UCSD

CENTER FOR MUSIC EXPERIMENT

COMPUTER

AUDIO

RESEARCH

LABORATORY

Startup Kit

CARL STARTUP KIT

Computer Audio Research Laboratory

Center for Music Experiment, Q-037
University of California, San Diego
La Jolla, California 92093

619 534 2230
619 534 4383
CARL - 4383
MUSIC - (3230)
GRAD STUDIES - 3552
THURS - FRI
Roberta wife
Sister's Irvine

Version Date: June, 1985

We have also our sound-houses, where we practice and demonstrate all sounds, and their generation. We have harmonies which you have not, of quarter-sounds, and lesser slides of sounds. Divers instruments of music likewise to you unknown, some sweeter than any you have; together with bells and rings that are dainty and sweet. We represent small sounds as great and deep; likewise great sounds extenuate and sharp; we make divers tremblings and warblings of sounds, which in their original are entire. We represent and imitate all articulate sounds and letters, and the voices and notes of beasts and birds. We have certain helps which set to the ear do further the hearing greatly. We have also divers strange and artificial echoes, reflecting the voice many times, and as it were tossing it: and some that give back the voice louder than it came; some shriller, and some deeper; yea, some rendering the voice differing in the letters or articulate sound from that they receive. We have also means to convey sounds in trunks and pipes, in strange lines and distances.

– from *New Atlantis*, Francis Bacon (1624).

CONTENTS

Introduction to the CARL Startup Kit

F. Richard Moore

Computer Audio Research Laboratory
Center for Music Experiment, Q-037
University of California, San Diego
La Jolla, California 92093

ABSTRACT

The Computer Audio Research Laboratory (CARL) is the computer music project at UCSD's Center for Music Experiment and Related Research (CME). The CARL Startup Kit is designed to provide a basic description of the major hardware and software facilities already operational at the CARL project. Beyond this general introduction, the CARL Startup Kit includes tutorial articles on

- the csound system for storing, retrieving, and manipulating soundfiles,
- the cmusic program for general-purpose digital music synthesis and signal processing, and
- the pvoc program for time-varying linear analysis and modified resynthesis of digital audio signals.

In addition, it includes

- a selected, annotated bibliography of relevant publications in the field of computer music,
- a few selected reprints, and
- a relatively complete listing and brief descriptions ("man" pages) of the software included in the CARL Software Distribution.

1. History

The Center for Music Experiment and Related Research (CME) was founded at UCSD in 1972 with major funding from the Rockefeller Foundation through the august efforts of its first director, UCSD Music Professor Roger Reynolds. Throughout the 1970's, as in almost every other field, computers played an ever-increasing role in musical research. This was reflected at CME in the work of such people as Ed Kobrin, who designed and built a minicomputer system to control the distribution of sound to multiple loudspeakers, and by the acquisition and development of a PDP-11/20 minicomputer system for initial exploration of digital sound synthesis. As a result of the second International Computer Music Conference held at UCSD in 1977, members of the UCSD Music Department resolved to embrace computer music as a major component of future music, and to create a major facility for its thorough investigation.

Viewed as a primarily artistic venture, founding support for the computer music project was provided jointly by the Rockefeller Foundation, the National Endowment for the Arts, and UCSD. The result of this effort was the establishment of the Computer Audio Research Laboratory at the Center for Music Experiment under the direction of the author, who left the research staff of Bell Laboratories in 1979 in order to found and develop the CARL project at UCSD.

The first system design issue was to decide on a working context.

One other major computer music system had been established in the mid-1970's at the Institute for Research and Coordination of Acoustics and Music (IRCAM) in Paris. The IRCAM system was based directly on the excellent Stanford Artificial Intelligence Laboratory (SAIL) computer system as augmented for musical work by John Chowning, James (Andy) Moorer, and others. By copying a well-working system, IRCAM researchers were able to attain state-of-the-art capabilities more or less immediately.

There were two disadvantages to the route chosen by IRCAM, however. One was that the computer hardware on which the SAIL system was based—the DEC PDP-10 architecture—was approaching the end of its technological lifespan. Computer technology changes so rapidly that estimates of the economic lifetime of computer hardware are on the order of only five years, after which time it becomes cheaper to replace a computer with newer technology than to continue maintaining the old one. Only software investment could justify the continuation of an older architecture, and the PDP-10 architecture was already over a decade old in 1979, making it ancient by computer standards.

The second disadvantage of copying a PDP-10 system architecture was its lack of general availability. Similar systems could be found, in fact, only at a handful of major universities—primarily those engaged in artificial intelligence research. This paucity of compatible systems made it difficult to share research results among centers, and virtually impossible to continue work on a project in one center that had been started elsewhere.

The first design criterion for the CARL system therefore became software portability based on a state-of-the-art computer architecture. This resulted in a system design that could withstand the vagaries of technological changes in the computer field—one which could make possible the unprecedented act of carrying work-in-progress from one computer music center to another.

In 1979, a most promising architecture was that of the "super minicomputer"—the DEC VAX-11/780. Similarly, the most widely available powerful operating system—in terms of the number of different types of hardware which could sustain it—was the Bell Laboratories UNIX system as extended by members of the computer science department at the University of California, Berkeley.

The VAX-11/780 computer running the Berkeley UNIX operating system first became operational at CARL in May, 1980.

The primary high-level programming language of the UNIX operating system is C. C represents a cunning compromise among the conflicting goals of power, elegance, efficiency, and portability. The success of C as a programming language is exemplified by the fact that well over ninety percent of the UNIX operating system itself is written in it.

Within the VAX/UNIX/C operating context, the next step was to determine what the system would do. Ideally, the solution to this problem is simple: the system should do whatever can be done with sound on a computer. Practically, however, the decision to build a system from the ground up for musical purposes required an implementation strategy that could not be executed instantaneously, meaning that certain

capabilities would not arrive as quickly as they might have if the system had been a straightforward clone.

The advantage of rethinking a computer music system from scratch was the designed-in coherence—from the user's viewpoint—of the resulting system.

Two major software packages were written first by necessity.

One was the csound system for the manipulation of soundfiles, written by Gareth Loy who joined the CARL staff in 1980 after completing his graduate studies in computer music composition and software development at Stanford University. The csound system allows multichannel sounds to be recorded in computer memory, played back, and edited in a variety of ways.

The second major software package was cmusic, written by the author. cmusic is a general purpose sound synthesis and processing program with an overall structure similar to that of the well-known MUSIC V program. Using MUSIC V as a model allowed—insofar as possible—cmusic to be apprehended and used immediatly by practitioners of computer music.

Refinement of the csound and cmusic packages continues to this day, and CARL software has grown to include many other programs as well. The CARL Software Distribution contains a "snapshot" of released CARL programs as they exist at the time the copy is made. No charge is made for this software, which represents many "person-years" of programming effort*. CARL software has already been sent to over a hundred sites worldwide, including MIT, Stanford, Lucasfilm, Bell Laboratories, UCSB, Cal Tech, and IRCAM, which now has a VAX/Berkeley UNIX system modeled on the one at CARL.

2. The CARL System Today

CARL facilities are used in support of research, production, and education in computer music. Major funding from the System Development Foundation has provided the financial support for a professional research staff, graduate student support, operating expenses, facilities maintenance, and a program of faculty residencies.

Current CARL research is focused on the creation of a multipurpose computer music workstation. In addition to running all CARL software, this workstation will include realtime hardware for performance acquisition and processing, and realtime digital audio synthesis and signal processing.

The program of faculty residencies involves the production of musical works using CARL facilities. For example, three works produced at CARL were featured at the New York Philharmonic Horizon '84 festival at Lincoln Center: "Transfigured Wind" by Roger Reynolds, "Bamboola Squared" by Charles Wuorinen, and "Towards the Midnight Sun" by Joji Yuasa.

CARL facilities are also used in support of graduate courses in computer music offered through the UCSD Department of Music. The CARL system user base currently numbers about 60 people, more than half of whom are graduate students engaged in the computer music course sequence, advanced seminars in computer music, and doctoral research on such topics as spatial manipulation of sound, natural sound analysis, computer-aided music composition, and digital sound synthesis techniques.

3. Getting Started

In order to pursue serious work in computer music one obviously has to know about music and about computers. The computer can—at least in principle—produce any sound that can come from a loudspeaker that the user knows how to describe. However, *what* one has to know about music is considerably augmented by the computer.

In order to make music with a computer, one must have a firm grasp of the same principles of music making that apply to traditional composition and performance. In addition, it is necessary to be able to translate this knowledge into precise statements about the nature of the sound to be generated by the computer. These statements arise from a musical intent which is integrated with an understanding of the physical properties of musical sound, an understanding of how that sound is perceived by listeners, and an understanding of how the physical properties of that sound may be described in terms of operations that a computer can execute.

* CARL software is available for a nominal administrative fee from CME.

If we make a list of the disciplines, subdisciplines, and interdisciplinary topics that contribute to the creation of computer music, writing each one on the page so that it is close to related topics, we arrive at a picture that looks something like Figure 1.

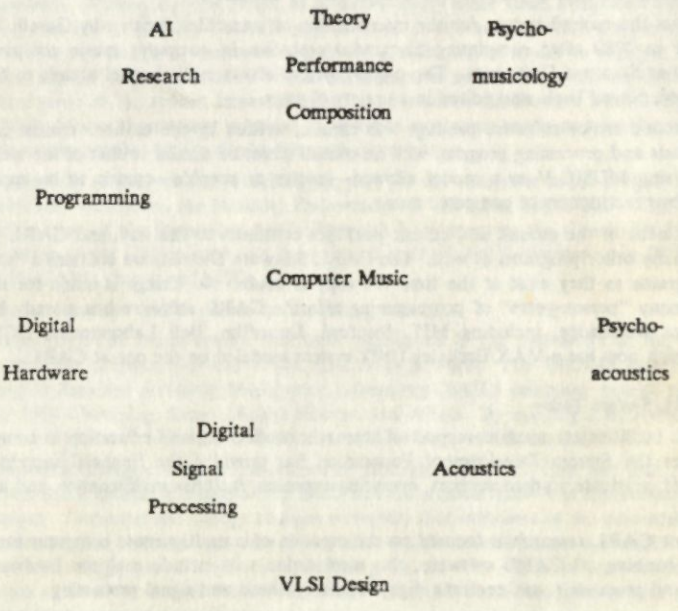


Figure 1: Proximity Graph of Fields Related to Computer Music

Figure 1 may be further organized into the five basic disciplinary areas that surround computer music: psychology, physics, engineering, computer science, and music. Within the field of music, the subfields of music theory, performance, and composition all apply directly to the creation of computer music. Within physics, the subfield of musical acoustics is directly relevant. Within engineering, the subfield of digital signal processing is germane. Within computer science, the subfield of computer programming is especially relevant.

Between these major disciplinary areas lie the interdisciplinary fields of psychomusicology, psychoacoustics, VLSI and electronic device design, digital hardware systems, and research in artificial intelligence.

The organization of Figure 1 may be made more explicit by including a five-sided figure pointing towards the five major disciplinary areas that surround computer music. The resulting picture is shown in Figure 2.

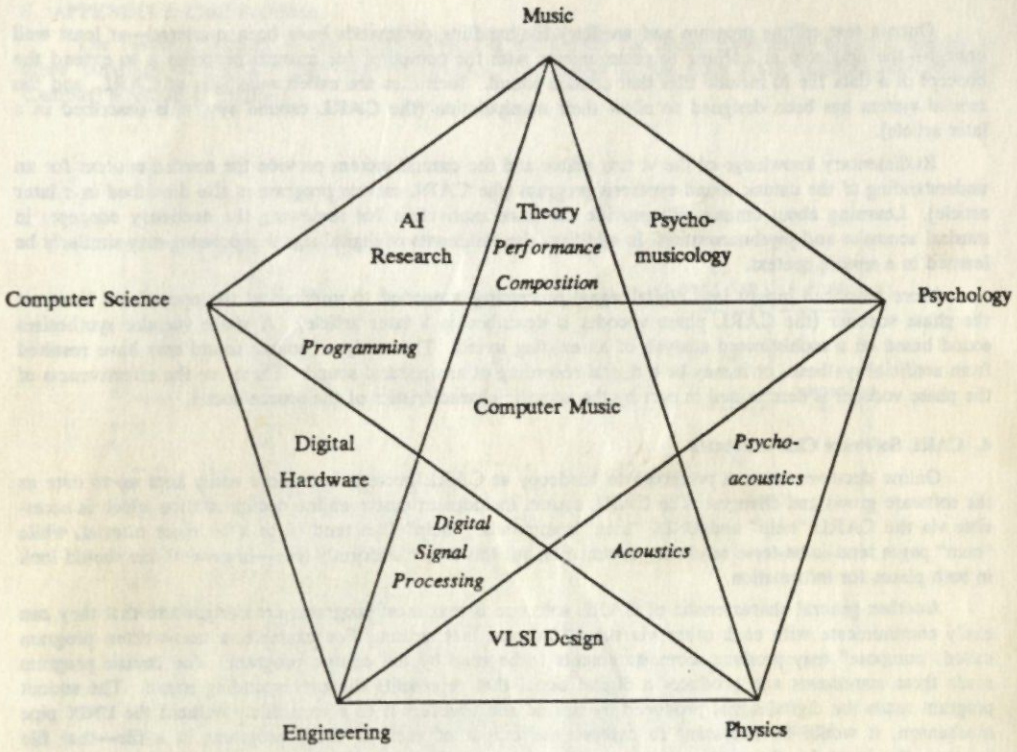


Figure 2: Disciplinary Organization of Computer Music

Selecting from this picture the elements most commonly applied to the creation of effective computer music (these are *italicized* in Figure 2) leads to the following list of essential prerequisites:

1) a mastery of the art and practice of music *composition* and *performance*,
2) an effective understanding of recent findings in musical *psychoacoustics*,
3) a basic understanding of musical *acoustics*,
4) an understanding of the fundamental principles of *digital signal processing*, and
5) computer literacy at the level of fluent *programming* in a high-level language such as C.

The study of computer music starts, then, with consideration of the relevant material from each of these five basic areas. Advanced research may require advanced understanding of a particular topic, but in any case a lack of effective knowledge in any of these five basic areas will eventually limit what a given user can accomplish.

Fortunately, it is not necessary to become expert in all of these subject areas before one can begin to explore the vast potential of computers as musical instruments. This startup kit contains a bibliography which points to much of the background information pertinent to engaging in serious computer music research.

For the beginner with a musical background, the first steps usually revolve around learning to communicate with the computer. The first major task beyond learning to log into and out of a computer system is to learn about file maintenance. The vi text editing program provides extensive facilities for entering and modifying text into computer files.

Once a text editing program and ancillary file-handling commands have been mastered—at least well enough—the next step in learning to communicate with the computer for musical purposes is to extend the concept of a data file to include files that contain sound. Such files are called *soundfiles* at CARL, and the csound system has been designed to allow their manipulation (the CARL csound system is described in a later article).

Rudimentary knowledge of the vi text editor and the csound system provide the needed context for an understanding of the cmusic sound synthesis program (the CARL cmusic program is also described in a later article). Learning about cmusic will provide the basic motivation for reviewing the necessary concepts in musical acoustics and psychoacoustics. In addition, the rudiments of digital signal processing may similarly be learned in a cmusic context.

More advanced insight into digital signal processing is needed to understand the operational limits of the phase vocoder (the CARL phase vocoder is described in a later article). A phase vocoder synthesizes sound based on a sophisticated analysis of an existing sound. The analyzed source sound may have resulted from artificial synthesis, or it may be a digital recording of any natural sound. Therefore the effectiveness of the phase vocoder is determined in part by the acoustic characteristics of the source sound.

4. CARL Software Characteristics

Online documentation is preferred to hardcopy at CARL because it is more easily kept up to date as the software grows and changes. The CARL system includes extensive online documentation which is accessible via the CARL "help" and UNIX "man" commands. "help" files tend to be a bit more tutorial, while "man" pages tend to be terse reference documents, but this is not uniformly true—in general one should look in both places for information.

Another general characteristic of CARL software is that most programs are designed so that they can easily communicate with each other via the UNIX pipe mechanism. For example, a user-written program called "compose" may produce score statements to be read by the cmusic program. The cmusic program reads these statements and produces a digital signal that represents the corresponding sound. The sndout program reads the digital signal produced by cmusic and transfers it to a soundfile. Without the UNIX pipe mechanism, it would be necessary to capture the output of each of these programs in a file—that file becomes the input for the next program in the sequence. With the UNIX pipe mechanism, however, one may execute the command

```
compose | cmusic | sndout
```

to effect virtual connections between the output of the compose program and the input of cmusic, and between the output of cmusic and the input of sndout. The data that flows from compose to cmusic is in the form of printable text statements. The data that flows from cmusic to sndout, however, is a digital signal, which can have three basic forms:

- a multiplexed stream of 32-bit binary floating point numbers,
- a multiplexed stream of 16-bit binary fixed point fractions, and
- a stream of printable (so-called ASCII) characters.

Further details about digital signal processing in the context of UNIX pipes is supplied in the reprint section of this startup kit.

5. A Final Word

A wise man once quipped that learning about computers provides one with many opportunities to exercise one's sense of humor. The mass of details always seems bewildering at first, as bewildering as a first stroll down a street where only Sanskrit is spoken. But computers and their associated languages are far simpler than "natural" languages, as evidenced by the fact that one can master computer languages much more quickly than, say, Russian or Greek, even though this difference may not seem self-evident at first.

Good luck, and welcome to CARL.

6. APPENDIX I: CME Facilities

The research context of the CARL project is the Center for Music Experiment which is housed in three buildings on UCSD's Earl Warren campus. The layout of the main CME building (408 Warren Campus) is shown in Figure 1.

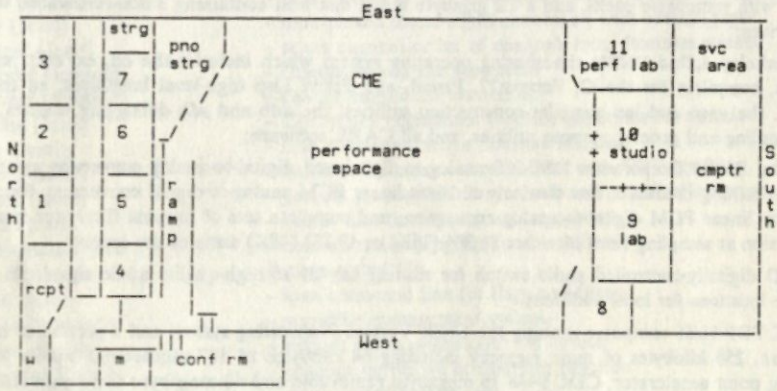


Figure 1: Floor Plan of the UCSD Center for Music Experiment

In addition to the numbered rooms which serve as offices for research staff, CME includes a performance space with movable acoustic panels and theater lighting system, a multipurpose conference room, and various laboratories and studios where technical facilities are housed.

CME facilities also include a quonset hut containing a woodworking shop and dry lab space, and a bungalow building which serves as a programmer workroom.

7. APPENDIX II: CARL Facilities

Present CARL facilities include:

- a DEC VAX-11/780 computer with 4 megabytes of main memory, a floating-point accelerator, two UNIBUS expansion cabinets for peripheral interface, about 50 lines for interactive terminals which are spread throughout the three CME buildings, about 6 dual-speed (300/1200 Baud) dialup modems including one with dialout capability, a Symbolics laser graphics printer, a TE-16 800/1600 bpi digital magnetic tape drive, an 80 megabyte RM03 disc drive with removable packs, two 300 megabyte CDC 9766 disc drives with removable packs, and a 1.2 gigabyte RA81 disc unit containing 3 non-removable Winchester disc drives;
- the Berkeley 4.2bsd UNIX† timesharing operating system which includes the ed, ex, edit, and vi text editors, compilers for the C, Fortran77, Pascal, and Franz Lisp high-level languages, an incremental loader, the yacc and lex compiler-construction utilities, the adb and sdb debugging utilities, over 200 file-handling and general purpose utilities, and all CARL software;
- a Digital Sound Corporation DSC-200 analog-to-digital and digital-to-analog conversion system with a buffer-chaining interface, two channels of 16-bit linear PCM analog-to-digital conversion, four channels of 16-bit linear PCM digital-to-analog conversion, and complete sets of lowpass filters for multichannel conversion at sampling rates of either 16,384 (16K) or 49,152 (48K) samples per second;
- a CMD digitally-controlled audio switch for routing any of 16 high-quality audio signals to any of 32 remote locations for local audition;
- a DEC PDP-11/55 computer running the UNIX Version 7 operating system and a portion of the CARL software, 256 kilobytes of main memory including 64 kilobytes of 167-nanosecond bipolar memory, a floating point accelerator, CDC 9448 16 megabyte removable and 48 megabyte non-removable disc cartridge drives, and an LPA-11 analog input/output unit;
- two Sun Microsystems, Inc. (SMI) SUN-1/50, one SUN-2/170, and one SUN-2/50 Motorola 68010-based computer workstations all running the SMI 4.2bsd operating system and CARL software via a 12-megabit Ethernet LAN which interconnects all main CARL computers;
- a digital electronics laboratory including an IBM-PC-compatible COMPAQ computer for laboratory instrument control, a 16-channel logic analyzer, an 80 megaHertz oscilloscope, a TI TMS320 emulator and hardware/software development system, Omnibyte 68KA and FORCE II single-board 68000 research computer systems, an Elite-II digital breadboarding system;
- a realtime digital performance laboratory including two YAMAHA DX-7 synthesizers, a YAMAHA TX-816 multiple synthesizer, several Roland MPU-401 computer-controlled MIDI interface, and a Roland MKB-1000 MIDI-compatible keyboard controller;
- a hybrid studio containing two SONY PCM-F1 and a SONY PCM-701 ES digital audio/videotape interface units, several ¼" and ½" reel-to-reel and cassette analog tape recorders, four channels of DBX noise reduction, analog mixing board, theater light control system, and patch bay;
- a CARL programmer workroom containing eight interactive terminals (one hardcopy) and quadraphonic playback;

† UNIX is a Trademark of Bell Laboratories.

8. APPENDIX III: CARL Software

The current CARL software includes the following programs, libraries, and documentation.

atob (1carl)	– ASCII to binary converter for floatsams
btoa (1carl)	– binary to ASCII converter for floatsams
burpsf (1csound)	– free space compaction for csound file system
cannon (1carl)	– continuous distribution function generator
catsf (1csound)	– concatenate sound files
cdsf (1csound)	– change csound file directory
cexpr (1carl)	– desk calculator with cmusic-compatible arithmetic functions
chan (1carl)	– multiplexed sound signal processing with parallel pipes
channel (1carl)	– select channel or list of channels from floatsam stream
chmodsf (1csound)	– change csound file protection
chownsf (1csound)	– change csound file ownership
chubby (1carl)	– Chebychev polynomial function generator for cmusic
click (1carl)	– find discontinuities in floatsam streams
cmpsig (1carl)	– multiple file/channel comparator with CRT graphics
cmusic (1carl)	– general purpose sound synthesis and modification program
comb (1carl)	– comb filter for piped sound signals
commandline (1carl)	– demonstration program for crack(3carl)
cpsf (1csound)	– move sound file
crack (3carl)	– scan command line for flags and files
csound (1csound)	– soundfile management system.
cspline (1carl)	– smooth curve interpolater for cmusic
derivative (1carl)	– produce derivative of floatsam function
dumpsf (1csound)	– dump sound files to tape
energy (1carl)	– plot mean square energy level of function
envanal (1carl)	– data reduction by hierarchical syntactic function analysis.
equtemp (1carl)	– list or match frequencies to equal tempered pitches
expr (3carl)	– cmusic-like expression evaluator
fc (1carl)	– floatsam counter
fftanal (1carl)	– analyze power spectrum of fft for peaks
filter (1carl)	– program for performing FIR or IIR digital filtering
fir (1carl)	– design finite impulse response digital filters
floatsav (3carl)	– iteratively save floatsams on a dynamic array
gain (1carl)	– scales a digital sound signal
gaindelay (1carl)	– scale and delay a digital signal
gen0 (1carl)	– normalize a function
gen1 (1carl)	– straight line segment function generator
gen2 (1carl)	– Fourier synthesis function generator
gen3 (1carl)	– simple line segment function generator
gen4 (1carl)	– exponential curve segment generator
gen5 (1carl)	– Fourier synthesis generator
gen6 (1carl)	– random table generator
genraw (1carl)	– read raw floating point functions into cmusic
getfloat, putfloat,	– floatsam input/output functions
glitch (1carl)	– one sample error detection
help (1carl)	– get help about CARL programs
hist (1carl)	– calculates histogram of sound signals
ichan (1carl)	– multiplexed sound signal processing with parallel pipes
impulse (1carl)	– generates an impulse signal
janus (1carl)	– smooth the attack and decay of a sound
libieee (3carl)	– standard ieee digital signal processing subroutines
libran (3carl)	– stochastic function library
libsf (3carl)	– subroutine library for csound file system.

locksf, unlocksf (lcsound) – lock/unlock a sound file system
lookfor (lcarl) – notify when named person logs in
lpc (lcarl) – linear predictive coding signal analysis
lprev (lcarl) – configurable sound reverberator
lsf (lcsound) – list sound files, sound file directories
m4n (lcarl) – read floatsams from M for N samples
mixsnd (lcarl) – csound file mixing program
mksfdir (lcsound) – make a sound file directory
mm (lcarl) – convert .m4 Makefile prototypes
mountsf, umountsf (lcsound) – mount and unmount a csound volume
mvsf (lcsound) – move sound file
noise (lcarl) – generate various flavors of random noise
nonzero (lcarl) – output non-zero floatsams
noteanal (lcarl) – note event parser for sound files
ochan (lcarl) – multiplexed sound signal processing with parallel pipes
offset (lcarl) – add constant offset to floatsam stream
opensf (lcsound) – user-level commands to open/close sound files.
para (lcarl) – parallel signal processing with common input & summed output
peak (lcarl) – calculates peak amplitude of sound signal
pianoroll (lcarl) – CARL score analysis program
play (lcsound) – play sound file(s) through DACs
polish (3carl) – convert mathematical expression to reverse polish notation
purgesf (lcsound) – nominate files to be purged
pvoc (lcarl) – phase vocoder
pwsf (lcsound) – print working sound file directory
quad (lcarl) – sound path interpreter for cmusic
r18 (lcarl) – read Stanford COPY format 18-bit sound sample tapes
readheader (lcarl) – read a header on a floatsam stream
reapsf (lcsound) – reclaim temporarily used sound file space
record (lcsound) – record sound file through analog-to-digital converters
rect (lcarl) – rectify floatsam stream
restorsf (lcsound) – restore sound files from magtape
retrosf (lcsound) – put retrograde of a sound file on standard output
reverb (lcarl) – one tap comb reverb
rms (lcarl) – finds the rms value of a digital sound signal
rmsf (lcsound) – remove sound file(s)
sched (lcarl) – find common hours for meeting times
scratchsf (lcsound) – holdsf – keepsf – set volatility status of files.
sdc (lcsound) – print map of soundfile disk cylinder usage
sfck (lcsound) – check sound file system for soundness
sfdt (lcsound) – print status of csound dump regimen
sfnorm (lcsound) – write normalized samples on standard output
show, fshow, yshow (lcarl) – CRT waveform display hack
signum (lcarl) – derive sign of floatsam stream
sndcmp (lcsound) – compare two sound files.
sndhist (lcsound) – produce histogram of sound file
sndin (lcsound) – read csound files onto standard output
sndout (lcsound) – write sound files.
sndpath (lcarl) – create/edit a sound trajectory
spect (lcarl) – apply FFT to floatsam stream
srconv (lcarl) – convert signal sampling rates by any positive rational number
step (lcarl) – step function generator for cmusic
stochist (lcarl) – plot histogram of stochastic function on CRT terminal
stripheader (lcarl) – remove a header from a floatsam stream

thresh (1carl)	– pass samples that exceed amplitude threshold
trans (3carl)	– generalized interpolation routine with transition parameter
ttyinfo (3carl)	– get tty info file entry
visf (1csound)	– edit sound file parameters
wave (1carl)	– generates simple test tones on the standard output
window (1carl)	– applies an envelope to a floatsam stream
wire (1carl)	– template program for inline digital sound signal processing
xform (1carl)	– transform sample data streams
zdelay (1carl)	– variable index interpolating delay

All CARL programs are documented either with a help file or a manual page, or both. Manual pages exist for all of the above programs.

Source code (in C) for CARL programs resides in a directory, typically containing a README file describing the steps needed to manipulate the program.

A device driver developed at CARL for the DSC-200 converters running under Berkeley UNIX is available to sites who submit their UNIX license number.

Introduction to the csound File System

D. Gareth Loy

Computer Audio Research Laboratory
Center for Music Experiment, Q-037
University of California, San Diego
La Jolla, California 92093

ABSTRACT

At least three distinct problems must be addressed in order to record, store, manipulate and play back high quality digital sound samples. The problems are 1) the sheer volume of data involved, 2) the high speed at which the data must travel and 3) the bookkeeping involved in managing the sound's vital statistics. Each of these problems goes beyond what can be handled conveniently by the regular UNIX† file system, so a special file system just for sound has been constructed, called the **csound** file system. This document covers most of what is needed to get started using it.

† UNIX is a trademark of Bell Laboratories.

1. In General

The **csound** file system has many points of similarity to the regular UNIX file system. They both are used to store and retrieve information from mass storage systems such as disks. But where the UNIX file system is suitable for storing most things, the **csound** file system has been developed to cover its weaknesses in sound sample storage. In many cases, the names of commands which manipulate **csound** files are synonyms to their UNIX counterparts. Where UNIX has an **ls** command to list the files in a directory, the **csound** file system has an **lsf** command ("list sound files"). Where UNIX has a **mv** command to move files, the **csound** file system has a **mvsf.** Similarly the UNIX **cp** command, which copies files, has a counterpart named **cpsf** in **csound.** Lastly, the UNIX **mkdir** program ("make a directory") becomes **mksfdir.**

From here the resemblance begins to fade. Whereas UNIX allows you a variety of ways to create new files, **csound** has but two: **sndout** and **record.** The **record** program creates a new sound file by reading the Analog to Digital Converters (ADC) and storing the samples in a sound file. **sndout** reads its standard input as the source of sound samples to be stored on the sound file system.

Similarly, there are two ways to get sound back from the **csound** file system: **sndin** and **play. play** writes samples from the **csound** file system to the Digital to Analog Converters (DAC), which allow a sound to be heard. **sndin** reads sound samples from the **csound** file system and writes them on its standard output. You can keep your sense of direction as to what "in" and "out" mean for **sndin** and **sndout** by remembering that the direction of transfer is always *with respect to yourself*. That is, **sndout** sends your samples *out* to the **csound** file system, **sndin** reads them back *in*.

2. Sndout

sndout reads its standard input and writes what it finds there to a file on the **csound** file system. In order to demonstrate **sndout** we must first have a program that writes samples for **sndout** to read. For example, at CARL there is a program called **wave** which produces a 1 second sine waveform, (sampled at 16KHz; sampling rates will be discussed further below). **wave** writes sound samples on its standard output suitable for input to **sndout.** We can pipe this output to **sndout** like this:

```
% wave |sndout waveform
```

This action causes the samples to be fed from **wave** to **sndout,** which sends them to the **csound** file system to be stored in a file called *waveform*. To be explicit, samples will be saved in a file whose full name is */some_filesystem/your_login_name/waveform* where *some_filesystem* is the name of a **csound** filesystem. For example, at CARL, there are currently three possibilities, for this, snda, sndb and sndc. You need only remember the *waveform* part when you are dealing with files you yourself have written. More on this subject later.

Now that the sound is stored in a file on the **csound** file system we can play it by saying:

```
% play waveform
```

(Note: **play** will only work for files stored on the **csound** file system. It will *not* work for files on the UNIX file system).

Let us say that after hearing the sound we decided to rescale its amplitude. A program to do this is called **gain.** We would use **sndin** to read the samples from the file *waveform,* then pipe them to **gain.** Lastly, we pipe it back to the **csound** file system to be saved.

```
% sndin waveform |gain .7 |sndout wave1
```

A couple of observations: **gain** has one argument which is a coefficient (in this case .7) with which to scale the signal on its standard input. Also, note that in writing out the file with **sndout,** we created a new file, *wave1,* instead of simply writing back to *waveform*. In general, *it is not possible to both read and write the same sound file at the same time*. While this is not strictly true under all circumstances, it is for the ones being described here. What would happen, if for instance we gave the command

```
% sndin foo |sndout foo
```

is that **sndout** would create a new file *foo* before **sndin** had a chance to read up the old one. The net result would be that the contents of the file would be lost.

If we really wanted this scaled-down version of the file to be called *waveform*, then we could use the command **mvsf** to "move" the sound to a file of a different name, e.g.:

```
% mvsf wave1 waveform
```

3. Sound File Allocation

Former releases of this software utilized a *fixed contiguous block* scheme of storage allocation which required knowing in advance of writing a sound file how much storage space would be required for it. It has been modified to use a *variable noncontiguous block* scheme which alleviates this requirement almost entirely. This means it is no longer necessary to specify sizes or durations, space is simply claimed as needed. One program, **record**, still uses the contiguous block method for reasons of efficiency.

4. About Samples

Sound is stored digitally as a stream of numbers called *samples*. Sound samples represent the instantaneous values of an acoustic waveform somewhat in the same fashion that successive frames of a moving picture store visual motion.

4.1. Sample Representation

There are three representations of samples:

- binary short integer (called *shortsams*),
- binary floating point, (called *floatsams*), and
- Arabic (you were expecting, perhaps, *jetsams?*).

Shortsam format is required by the DAC and ADC converters. It is in this format that samples are (usually) stored on the **csound** file system. *Floatsam* format is capable of a much wider dynamic range than *shortsam*, and is (usually) used by all CARL programs whenever programs pass samples between themselves via the standard input/output. *Arabic* is human-readable format, presented whenever a program notices that its standard output is connected to a terminal. Some programs refer to Arabic format as *ASCII*,* which simply means "in text format, suitable for printing on a screen".

Shortsams are 16-bit, two's complement, which means they have a range of integer values from -32768 to + 32767. The DACs convert these numbers to voltages between roughly -10 and + 10 volts. Each 16-bit sample occupies two 8-bit bytes on the disk. *Floatsams* have a very wide dynamic range (± 1.701411733192644270e38 on a VAX). However, only values in the *signed unit interval* (that is, values in the interval [-1,+ 1]) are (ordinarily) used to represent sound sample data. Floatsams are 32 bits, and occupy 4 bytes. All CARL programs which must convert from shortsam to floatsam sample formats equate floatsam -1.0 with shortsam -32767 and floatsam + 1.0 with shortsam + 32767. (Note that the value -32768 is not ordinarily used).

As already mentioned, samples are stored on disk as shortsams. But since **sndout**, reads its standard input for samples, it must therefore be reading floating point samples. Thus, one of the actions of **sndout** is to convert from floating point to integer before storing samples on the disk. Obversely, **sndin** must first convert shortsams to floatsams.

Programs can tell whether their standard input/output is connected to a terminal or a file or pipe. All CARL and **csound** programs which write sample data via their standard output first determine whether the output is a terminal. If so, the samples are printed in Arabic format suitable for display on a terminal, otherwise they are written as floatsams.

---

* ASCII stands for American Standard Code for Information Exchange.

4.2. Sample Frames

A *sample frame* is taken to mean one sample for each channel. For mono files, each sample is a sample frame. In Multi-channel files, the channels are stored in sample interleaved order. For instance, in a 4-channel file the individual samples are stored:

A,B,C,D, A,B,C,D, A,B,C,D, ...

The sample index i, which corresponds to a particular time in seconds T, and channel a, at sampling rate R, is given by

$$i = T*R*N + a$$

where N is the number of channels in a sample frame. The time corresponding to a particular sample frame index is

$$T = \frac{i}{R*N}$$

The number of channels is not limited by the **csound** file system. However, there is a limit placed on the number of channels that can be converted by the DACs and ADCs. The current limits at CARL are four channels of DAC and 2 channels of ADC.

4.3. Calculating Length of Files

Length of sound files in the **csound** file system is measured in units of *cylinders*. A *cylinder* is a large unit of storage on a disk, comprising roughly 300k to 400k bytes on disks used at CARL. Knowing the byte size of a sample and how many there are per second allows us to calculate the storage requirements of sound files of different duration. For instance, if we take a typical sample rate of 16384 samples per second, one second of mono shortsams requires

$$\frac{2\ bytes}{sample} * \frac{16384\ samples}{second} = 32768i\ bytes.$$

The actual cylinder size of one of the CARL disks (CDC 9766, mounted as /snd1) is 311296 bytes. Thus, a cylinder can store 311296/32768 = 9.5 seconds worth of shortsams at that sampling rate. The space requirement for stereo is exactly double that for mono, and for N channels, is N times the mono length.

4.4. Sample Rates

The *sample rate* is the number of sample frames sent to (or fetched from) the converters per second. It is possible to record a sound with the **record** program at nearly any sampling rate, from about 50 samples per second (sps) up to a maximum of 49152 sps*, often written as 48K**. At CARL, two sample rates are typically used, 48K sps for high-quality sound and 16K sps for tests and speech. If, as shown above 9.5 seconds can be stored at 16K sps, the amount of sound that can be stored on a cylinder at 48K sps goes down to 3.166... seconds. But the frequency range that can be represented goes up by a factor of three. For the 48K sps rate, the frequency response goes from 0Hz up to nearly 20KHz, which covers the range of human hearing pretty well. For the 16KHz rate, the frequency range is from 0Hz up to 6.5KHz.

There is obviously a tradeoff of sound quality vs. storage space needed. Since the sound file storage space is finite, and since one can get 3 times as much sound sampled at the lower rate to fit on a disk, there is a strong impetus to work most of the time at the slower sampling rate. Not only do files written at the 16 K sps sampling rate occupy smaller space, they also can be computed three times as fast. So here is a good rule of thumb: use the slow rate for tests or sketches, and the faster rate for

*Sampling rates may be different at different sites. At IRCAM for instance, the sampling rate is 48000 samples per second.

**The notation 48K is to be read as 48*1024 = 49152. At IRCAM, one could say 48k, which is 48*1000.

final products.

The sampling rate is analogous to the speed a tape recorder runs when recording a sound. Once recorded at some rate, playing back the sound at any other sampling rate merely changes the pitch and duration, not the sound quality nor the storage requirements. So the only places the sampling rate can be set is where sound files are created: *record* and *sndout*. For both programs, you supply a flag specifying the sampling rate you want. For instance:

```
% wave |sndout -R48K foo
```

will create sound file foo with the fast sampling rate.* If you do not specify the sampling rate, you get the default 16K sps rate.

5. Arithmetic Expressions In Flags

You may use arithmetic expressions in calculating numeric values written after flags. For instance, these three commands are equivalent:

```
% sndout -R49152
% sndout -R48K
% sndout -R"3*2^14"
```

In the second example, the **K** is a postoperator, which acts as a multiplier *times 1024*. Other postoperators include **k** which multiplies by 1000. Also, using the exponentiation operator, 3*2^14 = 49152. The binary operators +, -, *, and /, as well as () are also available.* Other postoperators which are available are **S**, for samples, **s** for seconds, **ms** for milliseconds, and **m** for minutes. These are useful for retrieving parts of a sound file, described next.

6. Sndin

sndin can read flags in addition to a file name to determine its operation. We'll focus here on begin time and an end time flags. For instance:

```
% sndin -b1 -e1.1 test
```

reads file test between times 1 and 1.1 seconds. That is, it starts with the 16384th sample (assuming the sampling rate is 16K), and writes out 1638 samples. If you say nothing, or if you simply supply the name of a file (as in our examples above), these times default to the beginning and end of the file. We could have asked it to calculate time in samples instead of seconds by using the S postoperator. For instance, to look at the first 200 sample frames*, regardless of sampling rate, we would say

```
% sndin -b0 -e200S
```

Postoperators are also cumulative; we could have said 200KS to read in the first 200*1024 samples.

It is often more convenient to specify a begin time and duration rather than a begin time and end time. To this end, the -b and -d flags behave as you would expect:

```
% sndin -b3 -d64S
```

reads starting at three seconds, and goes for 64 samples.

---

* **esound** programs understand 48K to mean 49152 by interpreting the 'K' as a *postoperator*. See below.

* Care must be exersized when using the binary operator * (for multiplication) and parenthesis. The UNIX shell will try to interpret them as part of a regular expression, and search for a matching file, usually with unsuccessful results which abort the command. It is necessary to enclose expressions with * and () in them in quote marks (") to avoid this.

*Note we said sample frames: if the file being read were stereo, we would read out 400 actual samples, two from each frame.

7. Lsf - Listing Sound Files

The **lsf** command, if given by itself, lists all the files in your current **csound** directory. It is possible to get a list of someone else's files simply by adding their name as an argument, preceded by a slash:

% lsf /frm

lists the names of frm's files. If you are interested in only whether a particular file exists, just name it:

% lsf /frm/joy

8. Manual Pages For Sound Programs

There is more information about a sound file than its name, which you can see by providing flags to **lsf** to modify its behavior. You can find out about these other flags and more about **lsf** itself, as well as all the rest of the programs discussed here, by reading their entries in the CARL section of the Unix Programmer's Manual. These manual pages give all the gory details too numerous to cover here. The manual exists in two forms: hardcopy and on-line, that is, in pressed wood pulp and on the computer. To read the on-line manual entry for **lsf** say:

% man lsf

The **man** command shows you a screenful at a time from the manual. After a full page of text is displayed on the screen it prints "-more-" at the bottom of the screen and waits for you to press the spacebar before showing the next screenful. If you would like to see all the sound file programs about which there are manual page entries, try another command:

% apropos csound

This will give you a list (it goes on for more than one screenful; ask someone to show you how to make it slow down) of all the sound programs in the manual. Each has its own manual entry available with the **man** command.

9. Cpsf - Copy Sound Files

This is a very simple program, similar to **mvsf.** But where **mvsf** simply changes the name of a sound file, **cpsf** physically copies the samples into another file. For example,

% cpsf source destination

where *source* is the name of the file to be copied, *destination* is where to put it.

10. Directories, and Filename Defaults

Just as the UNIX file system has a tree-structured directory, so does **csound.** When you log in, unless you change it, you will be set up to access **csound** files in your *home* **csound** directory. Your home directory may vary with the installation, and you may also change it with the program **cdsf.** All **csound** programs assume that an incomplete filename means to look in your home directory by default. For instance, when you just say **lsf** by itself this is the directory **lsf** examines. You can create subdirectories (branches) from this directory with **mksfdir.** For instance:

% mksfdir noises

creates directory /your_sound_filesystem/your_login_name/noises. (The euphemism "your_sound_filesystem" refers to the name of the sound file system you are assigned to when you got your login account. It varies with each system. At CARL, this will usually be "/snda".)

To use this directory, you must change your current sound file directory to be this new one. This is accomplished by the following statement:

% cdsf noises

Then saying

```
% ... lsndout fudge
```

will write a file /your_sound_filesystem/your_login_name/noises/fudge. This state of affairs will persist until you run **cdsf** again to change to a different directory. You can see that the directory you supply to **cdsf** determines where csound programs will subsequently look for, or create, sound files (so long as a sound file directory of that name exists). You can change to other user's directories. For instance:

```
% cdsf /his_sound_filesystem/frm
```

will cause the command

```
% sndin joy
```

to read file /his_sound_filesystem/frm/joy.

To change back to your own home csound directory, it suffices to say simply,

```
% cdsf
```

by itself.

If you need to refresh your memory as to what directory you are in, use the command **pwsf** which prints the current working sound file directory.

11. Relative Pathnames, Filenames

Sound file names used to be restricted to 10 characters or less; this has been expanded under 4.2BSD UNIX so that the length of names is virtually unlimited.

Unfortunately, csound supports no regular expression syntax. This means, for instance, that a command like

```
% lsf *
```

will not work. This is because the UNIX shell interprets the "*" before it is given to **lsf.** The shell interprets the "*" to refer to all UNIX files in the current UNIX directory, and can therefor not be used to indicate csound programs. There are clever ways to subvert this limitation, but they go beyond the scope of this document.

Relative pathnames are supported by all csound programs. That is, the pathnames "." and "..", and the pathname prefixes "./" and "../" are legal parts of sound file names, and operate exactly as they do in the UNIX file system.

12. Csound Programs

Here is a list of some of the currently available **csound** programs. The number in the first column is an indication as to the "relevance" the program has to beginning users, with 0 being important, and 9 being relatively unimportant. Some recent additions may be missing from this list.

0 lsf - list sound files, sound file directories
0 sndin - read csound files onto standard output
0 sndout - write sound files.
1 play - play sound file(s) through DACs
1 record - record sound file through ADCs
1 rmsf - remove sound file(s)
2 holdsf, keepsf - protect sound files from the purger
2 purgesf - purge over-ripe sound files
3 cpsf - move sound file
3 mvsf - move sound file
3 cdsf - change csound file directory
3 mksfdir - make a sound file directory
3 pwsf - print working sound file directory
4 sfck - check sound file system for soundness
4 sdc - print map of sound file block usage on a file system
5 catsf - concatenate sound files
5 sndcmp - compare two sound files.
5 sndhist - produce histogram of sound file
6 visf - edit sound file parameters
9 dumpsf - dump sound files to tape
9 locksf, unlocksf - lock/unlock a sound file system
9 opensf - closesf - user-level commands to open/close sound files.
9 burpsf - free space compaction for csound file system
9 restorsf - restore sound files from magtape
9 sfdt - print csound dump statistics

13. Epilogue

There, now you know more than you need to know about the sound file system. Go make some music.

The cmusic Sound Synthesis Program

F. Richard Moore

Computer Audio Research Laboratory
Center for Music Experiment, Q-037
University of California, San Diego
La Jolla, California 92093

ABSTRACT

cmusic is a general-purpose sound synthesis program developed by the author at the UCSD Computer Audio Research Laboratory (CARL). cmusic turns an acoustical description of a sound into a sequence of numbers (a digital signal) that represents the waveform of that sound. This numerical sequence may be stored in the computer, processed further by cmusic or other digital signal processing software, or passed to a digital-to-analog converter for conversion to analog form and eventual audition over loudspeakers. cmusic is modeled after the MUSIC V program developed by Max Mathews and others (including the author) at Bell Laboratories, but with many improvements and extensions. It is written entirely in the C programming language and runs (at CARL) under the 4.2bsd UNIX† operating system.

† UNIX is a Trademark of Bell Laboratories.

– CONTENTS –

1. Introduction

cmusic is a general-purpose computer program for making music. Basically, cmusic turns a written description of the physical characteristics of one or more sounds into a sequence of numbers called a *digital signal*. The written description is usually prepared by the cmusic user with a text editing program such as *vi*. If the description is stored in a file called *score.sc*, then the UNIX command

```
cmusic score.sc
```

will instruct cmusic to read and process that file—the resulting digital signal will be displayed as a (long) list of numbers on the user's terminal screen. The output of cmusic is typically fed to another program via the UNIX *pipe* mechanism, as shown in the following command.

```
cmusic score.sc | sndout musicfile
```

The sndout program stores the numbers in a *soundfile* which is named *musicfile* in the example. When its output is piped to another program, cmusic produces the signal in single-precision binary floating point form (rather than as ASCII text suitable for display on a terminal screen) for transmission efficiency. These binary numbers may require further conversion to fixed point binary fractions for digital-to-analog conversion.

Once cmusic has completed its task, the digital signal normally will be stored in a soundfile (*musicfile* in the previous example). We may then listen to the sound by issuing a command to specify that the digital signal stored in a soundfile is to be fed to the digital-to-analog conversion (DAC) system, as in the following example.

```
play musicfile
```

We might choose to record the sound while it is being converted.

1.1. Digital Signals

The numbers in a digital waveform are called *samples*. We may think of sample values as corresponding to instantaneous variations of air pressure around the mean atmospheric pressure at a point in space or as instantaneous positions of a loudspeaker cone as it vibrates back and forth to produce a sound. Sample values in a digital signal produced by cmusic are normally restricted to lie within the range ±1.0. Thus a "full amplitude" sine waveform as produced by cmusic might be represented by the sequence of numbers 0.0, 0.383, 0.707, 0.924, 1.0, 0.924, 0.707, 0.383, 0.0, –0.383, –0.707, –0.924, –1.0, –0.924, –0.707, –0.383, 0.0, 0.383, etc. We can construct a graph of such a digital waveform by erecting a sequence of vertical lines at equally-spaced positions along a horizontal axis representing time. The heights of the lines correspond to the values of the samples in the digital signal.

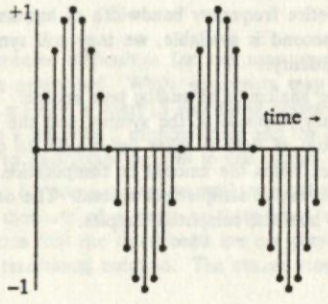


Figure 1: Graph of a Full-Amplitude Sine Waveform Produced by cmusic

1.2. Sampling Rates

Observing either the graph or the numbers themselves, we see that in this case 16 values are present in the first complete *cycle* of the waveform. One complete period of the sinewave therefore corresponds to the amount of time for 16 numbers to be processed by the DAC. If the DAC processes numbers at a rate of 1,600 per second, we can easily deduce that the frequency of the sinewave will be 100 cycles per second, or Hertz (Hz). If the DAC converts 16,000 numbers per second, however, the sinewave will be produced at a frequency of 1,000 Hz.

The rate at which the DAC processes samples is called the *sampling rate*. From the previous discussion we see that the relationship between the sampling rate and the frequency of a waveform is expressed by the formula

$$frequency = \frac{sampling\ rate}{samples\ per\ period} \quad (1)$$

In this simple case we could control the frequency of the output waveform by varying either the sampling rate or the number of samples in one period of the waveform. Since DACs for digital audio applications are usually designed to operate at one or at most a few fixed rates, frequency is normally controlled by varying the number of samples per period.

1.3. The Sampling Theorem

The basic rule of digital signals is called the sampling theorem (or sometimes the Nyquist theorem, after its inventor). For our purposes, the sampling theorem states that *in order to represent a sound containing frequencies up to X Hz as a digital signal, a sampling rate of at least 2X samples per second is required.* In effect, digital signals simply cannot represent frequency components greater than half the sampling rate. Any attempt to exceed this limit results in the phenomenon known as *foldover* or *aliasing*[1].

The sampling theorem describes one aspect of how much work cmusic must do in order to create musical sounds. If we wish to create a monophonic sound signal that includes frequencies up to 20,000 Hz (an optimistic estimate of the upper limit of human hearing), then the sampling theorem states that cmusic must generate *at least* 40,000 samples to describe *each second* of that sound. If more than one *sound channel* is in use, as in stereophonic or quadraphonic sound production, the sampling theorem applies independently to each channel. In other words, stereo requires twice the amount of data as mono, quad requires four times that amount, and so on.

The sampling theorem describes the minimum number of samples that are required for a given frequency bandwidth. In practice, it is usually necessary to use more than the theoretical minimum sampling rate because real DAC systems do not operate perfectly. In practical DAC systems, the frequency bandwidth is closer to 40% of the sampling rate than 50%.

It is not necessary to cover the entire frequency bandwidth of human hearing in all cases. If a sampling rate of, say, 16,000 samples per second is available, we may still synthesize frequencies up to about $0.4 \times 16{,}000 = 6{,}400$ Hz with excellent fidelity.

Sampling rates in digital audio are analogous to analog tape speeds. The faster the sampling rate (or tape speed) the better the high frequency response of the system, and the more expensive it is to operate. One aspect of this expense is the amount of time it takes for cmusic to compute the samples. All other things being equal, cmusic requires three times the amount of computation time to operate at a 48,000 Hz sampling rate than it would to operate at 16,000 samples per second. The other aspect of this expense is the amount of computer storage required to hold the computed samples.

---

1. For further information on the sampling process the interested reader is referred to "An Introduction to the Mathematics of Digital Signal Processing" by F. Richard Moore, in *Digital Audio Signal Processing: An Anthology*, William Kaufmann, Inc., 1985.

1.4. Computation Time

At this point one might ask a very reasonable question: Why not feed the samples directly from cmusic to the DAC system as they are computed, thereby eliminating the need for massive amounts of computer storage as well as the waiting time between preparing and listening to the score? The answer to this question is both simple and unfortunate. The DAC system must process the samples in *realtime*, that is, if the sampling rate is 50,000 Hz then the DAC for each output channel must process one sample every 1/50,000 second. Since one $50{,}000^{th}$ of a second is exactly 20 microseconds (millionths of a second, abbreviated μs), the question becomes: Can cmusic compute the value of one sample for each channel of sound output in just 20 μs? The answer is, in general, no.

A modern general-purpose computer such as a VAX-11/780 can perform arithmetic operations at a rate of about a million per second, so the amount of real time represented by one sample is sufficient for the computer to do about 20 operations. If we use cmusic to compute a stereophonic signal, then only about 10 operations could be executed on behalf of each channel before it would be time to move on to the next sample. If the method used to compute each sample requires more than a few arithmetic operations, then, it will be impossible for cmusic to operate in realtime.

Therefore cmusic normally operates in non-realtime. By gathering the sample values in a soundfile at whatever rate they can be produced, we eliminate the necessity to limit the complexity of the calculations used to obtain each sample. Once all the samples are computed, they can be processed in realtime by the DAC system.

The disadvantage of non-realtime operation is that it can (and often does) make things difficult for the cmusic user. Less obvious, perhaps, is the advantage of non-realtime operation: there is no particular limit on the complexity of operations that can be brought to bear on the synthesized sound. Realtime systems for sound synthesis always have a definite limit on computational complexity, meaning that some sounds simply cannot be produced by realtime music synthesizers, even digital ones. Additional complexity merely adds to the computation time that cmusic requires. No amount of complexity is necessarily impossible to achieve, although practical limits often are determined by the mean-time-to-failure of the computer system and the patience of the cmusic user.

The amount of time that cmusic will require to compute a digital signal is therefore determined by two factors: the sampling rate and the computational complexity involved in the production of each sample. The sampling rate determines both the available frequency range and the total number of samples that must be computed, while the computational complexity determines the amount of time needed to produce each sample.

Later we shall see that this complexity can vary over an enormous range. As a general rule of thumb, the "better" the resulting sound the more complex are the calculations needed to produce it. There are a few significant exceptions to this rule but alas, they are the exceptions that prove it.

1.5. Using cmusic

Non-realtime operation creates difficulties for the user, such as the need to specify everything in advance about the sound to be generated. While composers may be used to specifying musical scores in advance of their performance, no one is used to specifying performance in advance of its sound. Understanding the difference between a traditional musical score and the detailed behavior of the sound associated with its performance is the key to understanding how to use cmusic.

Imagine, for example, that for some reason we wish to synthesize a waltz. While composers traditionally notate the beats in every measure of a waltz as three even notes, even casual listening to any competently performed waltz confirms that the three beats are not played evenly. This is one way in which the sound of music differs from its traditional notation. The cmusic score for a waltz must describe the sound as the user wishes to hear it.

Many other examples of the difference between traditional music notation and the sound of music can be found. Consider musical dynamic markings such as *forte* (*f*) and *piano* (*p*). A trumpet playing *forte* not only produces a greater sound intensity, but also produces a different tone quality—or *timbre*—than a trumpet playing *piano*. Choosing a single timbre quality for a synthetic musical voice and varying only its intensity in order to suggest different musical dynamics will not usually sound very convincing in a musical context.

Neither do the sounds of traditional musical instruments remain uniform over the range of pitches that they can produce. Clarinets, for instance, have such distinct "registers" that they have descriptive names like *chalumeau*, *throat*, and *clarino*. The acoustical properties of low piano tones are vastly different from those of high piano tones.

This is not to imply that synthetic sounds must behave like natural ones. Many of the most interesting sounds that can be obtained via computer synthesis are precisely those that do *not* behave in a natural manner. But natural sounds have characteristic complexities that define degrees of sonic richness to which we are accustomed in music. Synthetic sounds that lack a similar degree of complexity may be easy to produce, but they are also immediately recognizable as synthetic, they quickly fatigue the ear, and they are generally less satisfying materials for musical expression.

cmusic can produce—in principle at least—any sound that can emanate from a loudspeaker. While loudspeakers themselves have important limitations as sources of sound, we know that they can produce satisfying likenesses of human speech, the sounds of traditional instruments playing either solo or in concert, and a host of other sounds both natural and artificial. All these sounds, then, can be synthesized by proper operation of cmusic.

Considerations up to this point, however, indicate real challenges in using cmusic to produce musically satisfying results. The initial challenge is to learn how to operate cmusic in order to obtain sounds. Vastly more important is the challenge to learn how sound operates in order to obtain music[2]. With respect to the latter challenge, computer programs are (so far) completely neutral. But cmusic does provide an enormously flexible tool that can be used to test theories, opinions, and guesses about sounds in general, and about musical sounds in particular.

1.6. A Concrete Example

A cmusic score is typically a text file prepared by the user with a program such as *vi*, *ex*, *ed*, or any other available text editor. If the scorefile is named as the last argument to the cmusic command (as in the command "cmusic score.sc"), cmusic will read and process that file. If no scorefile is named in the cmusic command, the cmusic program will read score statements from its *standard input*, or *stdin* as it is called in UNIX. For example, we might write a computer program that creates a cmusic score called "compose". We may link the output of the "compose" program with cmusic via a UNIX pipe command such as

```
compose | cmusic | sndout
```

In this example, the information that flows through the pipe from compose to cmusic consists of score (text) statements, while the information that flows from cmusic to sndout consists of binary floating point numbers constituting a *sample stream*, or digital signal. Similarly, the UNIX operating system allows us to run the compose and cmusic programs separately by first capturing the output of the compose program on a file with a command such as

```
compose > score2
```

We may wish to examine the *score2* file for correctness before combining it with other (partial) score files with a command such as

```
cat score1 score2 score3 | cmusic | sndout
```

In the last example, the score read by cmusic will consist of the *concatenation* of the information in file *score1*, followed by the information in file *score2*, followed by the information in file *score3*.

A cmusic score itself consists of cmusic score statements. Each statement in a cmusic score (with minor exceptions) must end with a semicolon (;) character.

cmusic score statements consist of a *command*, followed (usually) by an *action time*, followed by *arguments or parameters* required by the *command*, and ending with a *semicolon*. In general terms, then, a cmusic

2. For an excellent and authoritative survey of recent findings in musical acoustics and psychoacoustics, the interested reader is referred to *The Science of Musical Sound* by John R. Pierce, Scientific American Books, Inc., distributed by W. H. Freeman and Company, 1983.

statement has the following form:

command action_time parameters ;

The basic commands define either *instruments* that can produce sounds or *notes* to be played on one or more of these instruments. Other commands exist that allow the user to specify general characteristics of cmusic's operation (like setting the sampling rate to a particular value), and to let cmusic know that the score has ended.

As a concrete example, here is a complete (though rather simple) cmusic score.

```
1   #include <carl/cmusic.h>

2   instrument 0 simple ;                          {define instrument simple}
3           osc     b1 p5 p6 f1 d ;                {p5=amp, p6=freq}
4           out     b1 ;                           {monophonic output}
5   end ;

6   SINE(f1) ;                                     {define f1 as a sinewave}
7     {p1=cmd}   {p2=time}   {p3=ins}   {p4=dur}   {p5=amp}   {p6=freq}

8     note         0.00        simple      .10       0dB       A(0) ;
9     note         0.27        simple      .23      -6dB       A(0) ;
10    note         0.50        simple      .23      -6dB       B(0) ;
11    note         0.75        simple      .24      -6dB       Cs(1) ;
12    note         1.00        simple      .10       0dB       A(0) ;
13    note         1.25        simple      .10      -6dB       Cs(1) ;
14    note         1.50        simple      .45       0dB       B(0) ;
15    terminate ;
```

Example 1: A Simple cmusic Score (the line numbers are *not* part of the score—see text).

Example 1 includes line numbers that are *not* part of the cmusic score itself—they are present only so we can refer to them here.

Line 1 contains a special statement that sets up a host of predefined shorthand notations that may be used in the score.

Lines 2 through 5 define the simplest possible cmusic instrument. Line 2 says, in effect, that an instrument named *simple* is to be defined at time 0. In their simplest form, action times are given in seconds and refer to times with respect to the generated sound. Time 0, then, occurs before any sound is generated, and is often the action time for definitions that are needed before sound synthesis can begin.

Line 2 contains a *comment* enclosed in curly braces ({}). Curly braces may be used to delimit comments anywhere within a cmusic score. Whatever the user writes between curly braces is completely ignored by cmusic, i.e., comments are for human eyes only.

Line 3 contains the first statement in the definition of instrument *simple*. It specifies that an oscillator (i.e., an *osc unit generator*) should be used with its (signal) output connected to *b1*. *Unit generators* are computer subprograms contained in cmusic that perform various signal generating and processing functions.

b1 refers to an input/output (or simply i/o) block in cmusic parlance. An i/o block performs a function similar to that of a patch cord. By connecting the output of the *osc* unit generator to the input of the *out* unit generator mentioned on line 4 via i/o block *b1*, we effective "patch" the output of one unit generator to the input of another.

In the statement on line 3 we see the notation *p5* written as the second parameter following the *osc*

designator. The second parameter of the *osc* unit generator is an input that controls the *amplitude* of the generated waveform. The notation *p5* itself means "parameter 5," which refers to the fifth parameter of any *note* statement that plays this instrument.

The next parameter of an *osc* statement names the source of information that will control the *frequency* of the generated waveform. Here we see the notation *p6*, which refers to the sixth parameter of any *note* statement that plays this instrument.

The next parameter of an *osc* statement names the source of information that controls the *shape* of the generated waveform. Here we see the notation *f1*, which means that wavetable *f1* will be used to specify the shape of the output wave of this *osc*.

The final parameter, *d*, is a special notation that creates a computer memory location where the *osc* unit generator can keep track of what it is doing during the course of a note. In cmusic parlance it is called the *sum* location, which can be used to gain control over the phase of the generated waveform.

At this point the alert reader will have noticed that the *osc* unit generator can produce any (periodic) waveshape with an arbitrarily specifiable amplitude, frequency, and phase. The *osc* statement itself does not usually specify actual values for these parameters (although it could), but rather *sources of information* about these parameters that will be activated later on in the cmusic score.

Line 4 specifies that the output of the *osc* unit generator (*b1*) is to be output by cmusic. The *out* unit generator has only signal inputs. For monophonic sound synthesis, *out* has only a single input—whatever is connected to it will appear in cmusic's output signal. For multichannel synthesis, the *out* unit generator has as many inputs as there are channels—the signal fed into the first input will appear in output channel 1, the second input will be fed to channel 2, and so on.

Line 5 specifies the *end* of the instrument definition. At this point in the score we could define another instrument, or go on to other score specifications.

Line 6 specifies that wavetable *f1* should be set to a SINE waveshape. This is an example of a shorthand notation that is made possible by the statement on line 1 of the score. The shorthand notation

```
SINE(f1) ;
```

actually stands for

```
generate p2 gen5 f1 1 1 0 ;
```

This statement consists of a *generate* command followed by the necessary parameters to create a sine waveshape and to store that waveshape into wavetable *f1*. Details of this process will be discussed later.

The statements on lines 8–14 of the score are called the *notelist*. The notelist specifies notes to be played on instrument *simple*.

The first four parameters of every *note* statement always have the same significance. Parameter 1 (*p1*) is always the *note* command itself. *p2* is always the action time, which specifies the beginning time of the note, in seconds. *p3* is always the name of the instrument to be played. *p4* is always the duration of the note, again in units of seconds.

The remaining parameters of a *note* statement have no fixed meanings—their significance is determined by the definition of the instrument being played (i.e., the one named in *p3*). The meaning of *p5* and *p6* are the same for every *note* statement in this score, but that is because in this score only one instrument (*simple*) is played. Line 7 consists of comments that help identify the significance of the various parameters in the notelist.

p5, according to line 7, specifies the amplitude of the note, and *p6* specifies the note's frequency. We see that the amplitude value for each note is given with an indicator that the numerical value refers to a "dB"—or *decibel*—scale. Any decibel scale always requires a reference value—in cmusic the reference value is arbitrarily chosen to be 1.0. Therefore the notation "0dB" refers to an amplitude value of 1.0, while the notation "–6dB" refers to an amplitude value of approximately 0.5.

The last parameter of each note statement (*p6*) is a frequency, but the statement on line 1 allows us to use a shorthand notation. In this case the notation is rather peculiar, and it is definitely peculiar to cmusic. Rather than specifying frequencies directly in Hz (which is also possible in cmusic), tempered scale pitch

class names are given followed by octave numbers. In cmusic notation, the notation "A(0)" refers to *A440*. Adding 1.0 to the octave number raises the pitch by an octave, subtracting 1.0 from the octave number lowers the pitch by one octave. Sharps and flats are specified by appending "s" and "f" to the note names. Octave 0 is taken to be the octave starting at *middle C* and proceeding upwards on a piano keyboard to *B*. Therefore "Af(–1)" refers to *A-flat* a major third below *middle C*, "Cs(1)" refers to *C-sharp* a major third above *A440*, and so on. As you might suspect, the reference pitch can be easily modified for easy transposition and in any case all frequencies are available using "Hz" notation, i.e., we could just as easily have written "440Hz" instead of "A(0)", "493.883301Hz" instead of B(0), and so on.

The *terminate* statement on line 15 of Example 1 informs cmusic that the score is finished, i.e., this statement is the cmusic equivalent to a double barline.

1.7. Instrument Diagrams

While cmusic accepts instrument definitions only in text form as shown in Example 1, it is often convenient and instructive to depict instruments in diagrammatic form. Unit generators appear in this form as symbols similar to those used in block diagrams of electrical circuits.

The traditional symbol used for an oscillator unit generator such as *osc* is an elongated semicircle with inputs for amplitude and frequency and one output. It is similar to the standard electrical symbol for an *and* gate.

Figure 2: The cmusic *osc* Unit Generator

Here *amp* refers to an amplitude input value, *incr* refers to an *increment* input value that controls the frequency of the output signal, *wave* refers to a wavetable that controls the shape of the output waveform, and *out* refers to the output signal.

The *out* unit generator symbol is a circle with an input for each channel. The monophonic *out* generator is depicted as follows:

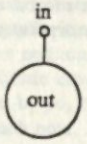


Figure 3: The cmusic *out* Unit Generator

We can now depict instrument *simple* from Example 1 with the following diagram.

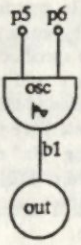


Figure 4: Instrument *simple* from Example 1

Figure 4 shows clearly the information flow in the instrument. *p5* controls the amplitude of the *osc*, *p6* controls its frequency, and the waveshape is depicted as sinusoidal. The output of the *osc* is fed via i/o block *b1* into an *out* unit generator. With a little practice, it is possible to translate fluently between such diagrammatic representations of cmusic instruments and their equivalent text form.

1.8. Improvements on Example 1

While the score shown in Example 1 would actually work if it were fed into cmusic (try it!), the sonic result would leave much to be desired (listen to it!). First of all the waveform has probably the dullest possible sound that has yet been invented. Even though some variation in amplitude and note timing is present, the performance sounds awkward and stilted. The individual notes do not have even an amplitude variation (*envelope*) over their durations—they just click on and off like a player Hammond organ with one drawbar pulled.

How could this score be improved? This simple question raises practically the entire question of how computer music should be made. The key to most answers to this question lies in the issue of *variation*. We now consider some basic techniques for achieving sonic variation using very simple means.

Sonic variation is both an art and a science. Recent findings in psychoacoustics can be of more than occasional guidance here. Ultimately, though, sonic variation with computers is as subtle—perhaps *more* subtle—than the art of musical performance itself.

1.8.1. Amplitude Variation

One of the most dunning atrocities committed against musical sensibility in Example 1 is the lack of dynamic amplitude variation. First of all the notes just click on and off—they have no characteristic envelope that changes over the course of a single note. A simple way to impose a characteristic rise and fall of amplitude over the course of a note is to use a second *osc* unit generator to generate an amplitude envelope for each note. Here is an instrument that incorporates such simple amplitude variation.

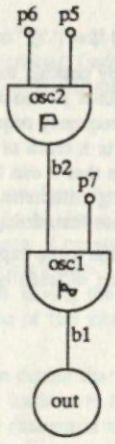


Figure 5: Instrument with Simple Amplitude Envelope Control

The instrument shown in Figure 5 uses two *osc* unit generators instead of one. *osc1* performs the same function as before—it produces the basic sound of the instrument. *osc2*, however, performs a completely different function: it controls the amplitude of *osc1* via i/o block *b2*. *osc2* also uses a completely different *kind* of waveform—one that is suited to producing a gradual rise in amplitude, followed by some sort of "pseudo-steady state", followed by a gradual decay back to zero amplitude.

Here is the corresponding instrument definition in cmusic form:

```
ins 0 simpenv ;
        osc     b2 p6 p5 f2 d ;          {p6=peakamp, p5=(1/dur)Hz}
        osc     b1 b2 p7 f1 d ;          {p7=carrier freq}
        out     b1 ;
end ;
SINE(f1) ;                               {carrier waveform}
ENV(f2) ;                                {amplitude envelope waveform}
```

Example 1a: A cmusic Instrument with Amplitude Envelope Control

Several changes may be observed in comparison with instrument *simple*. The main frequency of the main (*carrier*) oscillator is now controlled by *p7*—it operates exactly as before, but the note statements now have to specify pitch in a different parameter position. The amplitude input of the envelope *osc* is *p6*, which now controls the *peak amplitude* of the amplitude envelope. That is, the envelope for each note will start at zero, gradually build up to a value specified by *p6*, then die away at the end of the note, thus eliminating possible clicks at the beginning and end of each note.

p5 and *f2* as defined in instrument *simpenv* have no counterparts in instrument *simple*. In order to cause the envelope control oscillator to generate exactly *one period* of its waveform over the duration of a single note, we must set its frequency in a manner that depends on the duration of the note to be produced. Recalling that the frequency of a waveform is the reciprocal of one complete period of that waveform, we can deduce that the proper frequency for this application is just the reciprocal of the note duration expressed in Hz.

In order to accommodate instrument *simpenv*, the first *note* statement of the notelist can be modified as follows:

```
note 0.00 simpenv .10 1/p4Hz 0dB A(0) ;
```

We observe from this statement that it is fortunately possible to write an *expression* for *p5*—in this case one that refers to parameter *p4* (the note duration). Since *p4* has the value 0.1 in this statement, we calculate that *p5* must be set to 10Hz, which is exactly the frequency required to cause the envelope oscillator to generate exactly one complete period of its waveform in a tenth of a second. Of course we could have written 10Hz directly, but the advantage of the expression is that it can be exactly the same for all note statements in the score. Besides, computers are very good at doing arithmetic, and it is probably easier to specify a correct score by letting the computer do the arithmetic rather than doing it mentally.

It may seem a bit strange to include exactly the same expression on every note statement in the notelist. Could we avoid this by defining the instrument differently? For example, would the following definition for *simpenv* work?

```
ins 0 simpenv ;
        *osc    b2 p6 1/p4Hz f2 d ;          {*** A BAD EXAMPLE ***}
         osc    b1 b2 p7 f1 d ;              {p7=freq}
         out    b1 ;
end ;
```

Example 1b: An *Incorrect* Instrument Definition

Unfortunately, the answer is no for a very simple reason. While it is true that we always want to set the frequency control input of the envelope oscillator to "1/p4Hz", *this expression does not have the same value for every note statement!* Consider, for example, the value that *p5* would have in the second (modified) note statement. In fact, at the moment an instrument is being defined, *p4* has no meaningful value at all, so the meaning of "1/p4Hz" would be similarly undefined. cmusic would be forced to complain about such a definition, and the score would have to be rejected.

The final new element in our modified score is wavetable *f2*, which controls the *shape* of the amplitude envelope for each note. The details of this waveform obviously will have a strong effect on the nature of the sound produced by *simpenv*. The definition for *f2* given in Example 1a is "ENV(f2);", which is shorthand for the cmusic statement

```
generate p2 gen4 f2 0,0 -1 .1,1 -1 .8,.5 -1 1,0 ;
```

The graph of this general-purpose envelope function is at this point a bit more enlightening:

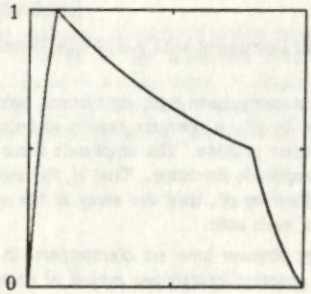


Figure 6: A Generic cmusic Amplitude Envelope Waveform (ENV)

We observe from Figure 6 that this function traverses an exponential curve from zero to its maximum value in one-tenth of its total duration. It then sags—again exponentially—from its maximum value to one-half of its maximum value when 80% of its duration is expended, then it falls exponentially to zero.

In our *simpenv* instrument, this function[3] (*f2*) will have its maximum value determined by *p6*—the "peak" amplitude—and its total duration determined (indirectly) by *p4*—the note duration. Each different combination of *p4* and *p6* will yield a (slightly) different amplitude envelope, and some measure of amplitude variation will have been achieved.

1.8.2. Frequency Variation

Even with the addition of an amplitude envelope, the sound specified by the score as we have modified it will be quite unmusical. We might decide by listening that the pitches are unnaturally steady and unwavering throughout the duration of each note. One way to combat this lack of sonic richness could then be to introduce a slight vibrato in the pitch of each note. Again we can use other note-to-note variations to control the parameters of this vibrato, such as having the note duration affect the vibrato rate and depth. We even have the option to control the shape of the vibrato, but to keep things simple let's use a simple sinusoidal shape.

A little musical intuition might lead us to decide that the longer the duration of a note, the slower and deeper (i.e., wider) its vibrato should be. This leads us to the interesting and instructive problem of deciding how to associate changes in note duration with changes in vibrato rate and depth.

First of all, we need to decide on the extremal limits for both rate and depth. *How* fast are slow and fast vibratos? *How* deep are shallow and deep vibratos?

Such questions pervade the creation of computer music, and they all share one important property: they are opportunities for research. One type of research involves examining the literature on vibrato. Surely this is not the first time such questions have been asked! The second type of research is empirical. This usually involves synthesizing a few or many test tones in which vibrato rate and depth are varied in a systematic way and listening to the results. Computers are excellent tools for turning guesses about sound into sonic experiences, and many times we find that examination of the literature leaves us no choice anyhow but to experiment.

For the sake of brevity, let us suppose that we have done some experimentation and concluded that a slow vibrato vibrates at about 3 Hz, and that a really fast one wiggles along at about 10 Hz. Further, let us suppose that a shallow vibrato needs to be on the order of one-eighth semitone in order to be audible, and that a reasonably wide one might encompass as much as a whole tone, aging opera singers to the contrary notwithstanding. We now need to examine the range of durations in our score in order to decide on a correspondence between them and our vibrato ranges.

The first few notes of our American folk tune have durations that range from 0.1 to 0.45 seconds. Following our previous prescription, we assign (arbitrarily) a correspondence between a 0.1 second duration with a vibrato rate of 10 Hz at a width of one-eighth semitone. Similarly, we wish a duration of 0.45 second to be associated with a vibrato rate of 3 Hz at a total width of about a whole tone. Intermediate values of duration should yield intermediate values for vibrato rate and depth.

The vibrato depth will also depend on the pitch of the note, for the size of a whole step is not the same at all frequencies. Conjuring up our knowledge of tempered tuning, we recall that the size of a tempered semitone is a frequency ratio equal to the twelfth root of two, which is about 1.059265. Multiplying this number by itself twelve times yields 2.0 (by definition), which is the frequency ratio of an octave. Multiplying any frequency by this number yields the frequency of the tone one semitone higher, and dividing by this ratio yields the frequency of a pitch one semitone lower. A total vibrato width of a whole tone would correspond to a vibrato amplitude of plus or minus a semitone referenced to the frequency of the note. Similarly, a total vibrato width of an eight-tone would correspond to a vibrato amplitude of a sixteenth tone.

Wouldn't it be nice if cmusic could help with the calculations? Fortunately, it can. cmusic expressions include not only normal addition, subtraction, multiplication, and division operations, but also the ability to raise any number to any power, as well as a host of others. In cmusic notation the expression "2^(1/12)" stands for the twelfth root of two, just as in the mathematical expressions $2^{1/12}$ and $\sqrt[12]{2}$. Thus the vibrato amplitude (Δf) for a note duration of 0.45 seconds will be approximately the difference between the main

3. For historical reasons the terms *wavetable* and *function* (or *stored function*) are used synonymously in cmusic.

frequency of the note and $2^{1/12}$ times the main frequency of the note, while Δf should be eight times smaller than this, or the difference between the note frequency and $2^{1/96}$ times the note frequency, for a note duration of 0.1 second.

A few moments cogitation with paper and pencil in hand demonstrates how to map a variable that varies over one range into another variable that varies over another range. Suppose x varies between x_1 and x_2 and y varies between y_1 and y_2. If we wish to establish a correspondence (mapping) between x and y so that when x lies p percent of the way from x_1 to x_2 then the corresponding y will lie p percent of the way from y_1 to y_2, we can use the following general relation:

$$y = \frac{x-x_1}{x_2-x_1} \times (y_2-y_1) + y_1 \qquad (2)$$

We want to establish the following correspondence between the note duration (*p4*) and vibrato amplitude.

duration		vibrato amplitude (Δf)
0.1	→	*note frequency* $\times 2^{1/96}$ – *note frequency* Hz
0.45	→	*note frequency* $\times 2^{1/12}$ – *note frequency* Hz

The desired vibrato amplitude (Δf) as a function of *p4* is then given by the somewhat quaint relation

$$\Delta f = \textit{note frequency} \times \left(2^{\frac{p4-0.1}{0.45-0.1} \times \left(\frac{1}{12}-\frac{1}{96}\right)+\frac{1}{96}} - 1\right) = \textit{note frequency} \times \left(2^{0.20833 \times p4 - 0.010416} - 1\right) \qquad (3)$$

In the first form, we see that the exponent of 2 goes from 1/96 to 1/12 as *p4* goes from 0.1 to 0.45, as desired (the second form is a "simplified" but less enlightening version of the first).

We want our second mapping function to establish the correspondence:

duration		vibrato rate (f_{vib})
0.1	→	10 Hz
0.45	→	3 Hz

which results in the following relation:

$$f_{vib} = 10 - \frac{p4-.1}{.45-.1} \times (10-3) = 9.714286 - 20 \times p4 \qquad (4)$$

Given these vibrato/duration correspondence functions, we now turn to their implementation in cmusic. Up to now we have been controlling the frequency of the main oscillator directly with a note parameter. In order to obtain vibrato, the frequency of the main oscillator must be a time-varying signal that wavers around a central pitch for each note. Stated mathematically, we want to change the frequency of the main oscillator from a constant value f to a time-varying value $f + \Delta f \times \sin(2\pi f_{vib} t)$, where Δf is the vibrato amplitude and f_{vib} is the vibrato rate. Expressed graphically, we are changing the main oscillator frequency from a steady, non-time-varying quantity that looks like this:

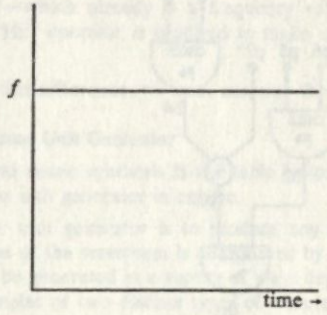


Figure 7: Graph of a Steady Frequency (f)

to a time-varying frequency that looks like this:

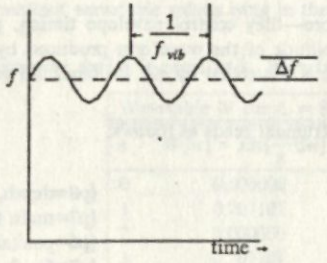


Figure 8: Graph of a Vibrato Control Function $f + \Delta f \times \sin(2\pi f_{vib} t)$

This can be accomplished in cmusic through the use of an *adn* unit generator. The *adn* unit generator simply adds all of its inputs to form its output. The inputs may be any combination of constant values and time-varying signals. Since the number of inputs is arbitrary, there may in general be *n* of them—hence the name *adn*.

A cmusic instrument capable of producing vibrato as well as amplitude envelope control may be diagramed as follows:

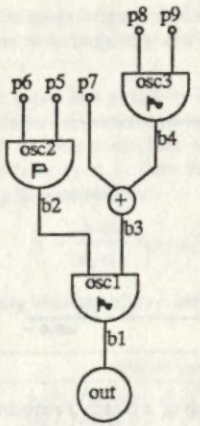


Figure 9: cmusic Instrument with Vibrato and Amplitude Control

p5, *p6* and *p7* are defined as before—they control envelope timing, peak amplitude, and main frequency respectively. *p8* controls the amplitude of the waveform produced by *osc3*, hence it will control the vibrato amplitude (Δf). The vibrato rate is controlled by *p9*. I/o block *b3* will contain precisely the vibrato control function shown in Figure 8.

A cmusic score that defines this instrument reads as follows:

```
ins 0 vib ;
        osc     b4 p8 p9 f1 d ;          {p8=depth, p9=rate}
        adn     b3 p7 b4 ;               {p7=main freq}
        osc     b2 p6 p5 f2 d ;          {p6=peakamp, p5=(1/dur)Hz}
        osc     b1 b2 b3 f1 d ;          {p7=freq}
        out     b1 ;
end ;
SINE(f1) ;                               {carrier waveform}
ENV(f2) ;                                {amplitude envelope waveform}
```

Example 1c: cmusic Instrument with Amplitude Envelope and Vibrato

Notice that it is permissible to use a wavetable more than once—*f1* (the sine wavetable) acts as a main waveshape and as a vibrato waveshape in instrument *vib*. The *adn* unit generator statement specifies that *b3* is the sum of *b4* and *p7*, thereby producing the time-varying frequency control function needed to control the output oscillator.

Incorporating the duration/vibrato correspondences worked out previously, the first *note* statement might now look like this:

```
{p1}  {p2}  {p3}  {p4}   {p5}    {p6}  {p7}      {p8}                          {p9}
note   0     vib    .1   1/p4Hz   0dB   A(0)   p7*(2^(0.20833*p4-0.010416)-1)   9.714286-20*p4Hz ;
```

Following programming language conventions, the asterisk (*) means *multiply* in cmusic. Unlike programming languages, though, cmusic requires arithmetic expressions to be written *without intervening blank spaces*—blanks are used to separate expressions from each other[4].

4. Actually, any combination of commas with whitespace may be used to separate fields in cmusic statements. Commas can sometimes be used to enhance the clarity of field organization.

Note finally that the expression for *p9* contains a "Hz" operator while the expression for *p8* does not. Since *p8* is derived from *p7*—which already is a frequency value—no "Hz" operator is required. *p9* is derived from *p4* only, so a "Hz" operator *is* required to make cmusic treat the resulting number as a frequency value.

In order to understand this difference, we must examine the *osc* unit generator in more detail.

1.8.3. Operation of the Oscillator Unit Generator

The workhorse of digital music synthesis is the table lookup oscillator. The basic form of this sonic cornucopia is known as the *osc* unit generator in cmusic.

The purpose of the *osc* unit generator is to produce any periodic waveform at any amplitude, frequency, and phase. The shape of the waveform is determined by a list of numerical values stored in a *wavetable*. Wavetable values may be generated in a variety of ways depending on how the *osc* unit generator is to be used. We have seen examples of two distinct types of wavetables—one that holds a sine waveform and one that holds the shape of an amplitude control function.

Suppose a wavetable, W, contains a sequence of L numbers $W[0], W[1], \cdots, W[L-1]$ and that these numbers represent the shape of one period of a periodic wave such as *sine*. Since $\sin(\theta)$ goes through one cycle as θ goes from 0 to 2π, $W[0]$ would be equal to $sin(0)$, $W[1]$ would equal $\sin(2\pi/L)$, and $W[L-1]$ (the last entry) would have the value $\sin([L-1]\times 2\pi/L)$.

The maximum value of any number in wavetable W will be ± 1 since the *sine* function itself oscillates in this range. Obviously we could multiply every value in the table by an arbitrary amplitude A. cmusic conventions generally rely on *normalized* wavetable values lying in the range ± 1—amplitude scaling is performed by unit generators like *osc*.

As a concrete example, suppose that L is equal to 8. Wavetable W then would contain the numbers:

Wavetable W for $L = 8$	
n	$W[n] = sin(\frac{n}{8} \cdot 2\pi)$
0	0.000000
1	0.707107
2	1.000000
3	0.707107
4	–0.000000
5	–0.707107
6	–1.000000
7	–0.707107

Repeatedly outputing the numbers directly from wavetable W would result in a digital signal that looks like this:

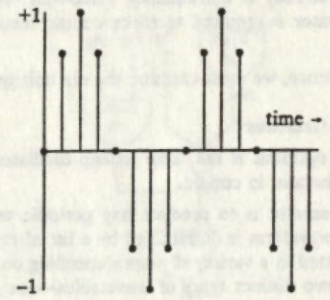


Figure 10: Cyclic Graph of Wavetable *W*

The frequency of the output wave would be R/L Hz, where R is the sampling rate and L is the table length. In this case the table length would be the same as the length of one period of the output waveform.

Suppose now that *every other number* from wavetable *W* is chosen for output. The resulting digital signal would look like this:

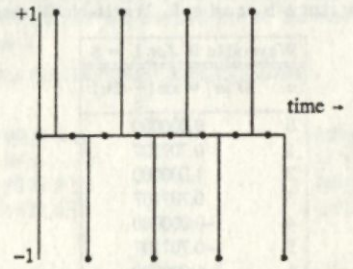


Figure 11: Cyclic Graph of Even-Numbered Entries from Wavetable *W*

Figure 11 shows a graph of the numbers $W[0], W[2], W[4], W[6], W[0], W[2], \cdots$. This waveform obviously has a frequency *twice* that obtained in the first case, since the period of each cycle is halved.

We could write a computer program to generate this sequence by setting up a variable n and incrementing it repeatedly by 2, starting at $n=0$. Before incrementing n each time we choose wavetable entry $W[n \bmod L]$ for output[5] ($L=8$ in this case). Modulus arithmetic is used to obtain the *cyclic* behavior needed to index the table. The *increment* applied to n determines how far we skip along in the table at each step, thereby determining the period of the generated waveform.

The waveform shown in Figure 10 results from incrementing n by 1 at each step (thereby cycling through *every* number in the table). The Figure 11 waveform results from incrementing n by 2 each time, causing the output waveform to have twice the frequency as before. We deduce, then, that *the increment value is directly proportional to the frequency of the generated waveform.*

Combining the direct proportionality of frequency to sampling rate and increment with the inverse

5. *a mod b* (read "*a modulo b*") is defined as the *remainder* that results when *a* is divided by *b*. For example, *a mod 10* will be the last decimal digit in *a*, *a mod 2* will be 1 is *a* is odd and 0 if *a* is even, etc.

proportionality of frequency to table length, we obtain the relation

$$frequency = \frac{increment \times sampling\ rate}{table\ length} \tag{5}$$

or, in terse mathematical notation,

$$f = i \times \frac{R}{L} \tag{6}$$

Solving this relation for increment in terms of the other values results in the relation

$$increment = \frac{frequency \times table\ length}{sampling\ rate} \tag{7}$$

or, in terser notation:

$$i = f \times \frac{L}{R} \tag{8}$$

For given values of table length and sampling rate, Equation (8) predicts the increment needed to obtain a particular frequency in the output waveform.

Only one conceptual problem remains: What if the predicted increment value is not an integer? Suppose, for example, that R=10,000 and L=100. To obtain an output frequency of 100 Hz, Equation (8) states that the increment value must be 1. To obtain 200 Hz, the increment would have to be 2. What if we wanted to generate 150 Hz? Equation (8) states that the necessary increment would be 1.5. How could we choose every 1.5th value from the wavetable, i.e., how could we generate the sequence $W[0], W[1.5], W[3], W[4.5], \cdots$?

We can obtain the necessary sequence in either of two basic ways: by *truncation* or by *interpolation*. Truncation involves dropping the fractional part of a number, so that any wavetable index between an integer I and $I+1$ would be interpreted as I. A truncating table lookup oscillator would output the sequence $W[0], W[1], W[3], W[4], \cdots$ as an approximation to the desired sequence $W[0], W[1.5], W[3], W[4.5], \cdots$.

Truncation is simple and fast. At the cost of more computation, however, interpolation between $W[I]$ and $W[I+1]$ by any of several means can be used, the simplest of which is *linear* interpolation. Assuming that the wavetable index lies p percent of the way from I to $I+1$, linear interpolation involves computing the corresponding value that lies p percent of the way between $W[I]$ and $W[I+1]$. A linearly interpolating table lookup oscillator would output the sequence $W[0], W[1]+.5\cdot(W[2]-W[1]), W[3], W[4]+.5\cdot(W[5]-W[4]), \cdots$ as an approximation to the desired sequence $W[0], W[1.5], W[3], W[4.5], \cdots$.

Provided the wavetable is not too short, either method can be used successfully. Both methods result in some distortion of the shape of the generated wave, but all other things being equal, the distortion is smaller for interpolation than for truncation. For a sinusoidal waveshape, the distortion is about *16 times* smaller for interpolation than for truncation. The wavetable for an interpolating oscillator therefore can be about 16 times smaller than for a truncating oscillator for equivalent distortion, at least for sinusoidal waveforms.

cmusic has both truncating and interpolating oscillator unit generators called *osc* and *iosc* respectively. *iosc* runs about three times slower than *osc*, but *iosc* also needs a wavetable 16 times smaller than *osc* to achieve the same distortion. Conversely, for a given size of wavetable, *iosc* produces a waveform with 16 times less distortion than *osc*. Why, one might wonder, would anyone ever use *osc*?

The answer is that computational efficiency is often more important than distortion, especially if that distortion is negligible. If low distortion is important then speedy results may be obtained *by increasing the wavetable size* (provided the computer has enough memory) rather than resorting to interpolating oscillators[6].

Unless otherwise specified, cmusic will use a table size of 1024 (=L), which is sufficient to allow use of the efficient *osc* unit generator in all but distortion-critical situations. A very low-distortion sinewave, such as might be required for a precise psychoacoustic experiment, could be generated by increasing the wavetable

6. For further information on such distortion the interested reader is referred to "Table Lookup Noise for Sinusoidal Digital Oscillators" by F. Richard Moore, in *Foundations of Computer Music*, MIT Press, 1985.

size, by using *iosc*, or both.

Returning to the operation of the *osc* unit generator, we now can appreciate an exact description of its algorithm. The n^{th} sample of output, O_n, is calculated in a two-step process:

$$O_n = A_n \times W[\lfloor S_n \bmod L \rfloor]$$

$$S_{n+1} = S_n + I_n$$

Example 2: Operation of the cmusic *osc* Unit Generator

where:

O_n is the n^{th} *output* sample,

A_n is the *amplitude* at sample n,

I_n is the wavetable *increment* at sample n,

W is a *wavetable* of length L,

S_n is the *sum* of increment values up to the n^{th} sample, and

$\lfloor x \rfloor$ is the *floor* function of x, i.e., the value of x truncated to its integer part.

Note the correspondence between the oscillator algorithm (above), the general form of the *osc* unit generator statement,

osc output amplitude increment wavetable sum ;

and the symbol for the oscillator unit generator (in which mention of the *sum* parameter is customarily omitted).

In typical uses of *osc* such as the following statements taken from Example 1c,

```
osc     b4 p8 p9 f1 d ;
osc     b2 p6 p5 f2 d ;
osc     b1 b2 b3 f1 d ;
```

we see that

- *osc* outputs are (always) of type *b* (signal),
- *amplitude* and *increment* inputs are usually of type *b* (signal) or *p* (constant),
- the *wavetable* is typically of type *f* (table), and
- the *sum* is of type *d* (dynamic variable).

Dynamic variables are used wherever possible in cmusic to hold state information for unit generators[7]. Dynamic variables are memory locations that are created automatically by cmusic at the beginning of a note.

7. *State information* refers to information used to keep track of where the unit generator is in a sequence of operations, such as the *sum* variable in the oscillator.

The purpose of this procedure is to allow cmusic instruments to play *any number of notes simultaneously*! State variables appear explicitly in unit generator statements (rather than being handled invisibly) to allow user access to them—the *osc sum* variable, for example, may be initialized to a non-zero value at the beginning of a note to obtain an initial phase offset in the generated waveform.

Now it is possible to fully understand the "Hz" notation used in cmusic. "Hz" and "dB" are examples of cmusic *postoperators*. Their effect on the quantity being calculated normally occurs last, after all addition, subtraction, exponentiation, etc., have been finished (parentheses may be used to alter the normal order of operations, of course).

The effect of the "Hz" postoperator is to multiply the value of an expression by the prevailing default table length for wavetables (L), then to divide the result by the prevailing sampling rate (R). In other words, if "ξ" is any cmusic expression, then "ξHz" is also a cmusic expression that stands for $(\xi)\times L/R$. Equation (8) confirms that this will convert the value represented by ξ to the corresponding *increment value* needed to make the oscillator generate its waveform at ξ Hz. This may seem like a roundabout way to do things, but it allows the expression "440Hz" at the increment input of an oscillator to specify *A440* as a result.

1.8.4. Timbre Variation

Amplitude and pitch variation are essential components that contribute to the richness of a synthetic musical sound. So far we have seen examples of how such variations might be applied to a simple sinusoidal waveform. Amplitude and pitch variation contribute strongly to the musical quality of sinusoids for a simple reason: a sinewave that varies in amplitude and pitch is no longer a "pure" sinewave at all!

A true, "pure" sine waveform is the one represented by the mathematical expression

$$f_1(t) = A\sin(\omega t + \phi) \tag{9}$$

where:

- t represents (continuous) time,
- A is a constant amplitude in arbitrary units,
- ω is radian frequency, i.e., $\omega = 2\pi f$, where f is a frequency in units of Hz, and
- ϕ is an arbitrary constant phase offset.

Equation (9) describes a "pure" sinewave, which exists for all values of t (i.e., for all time, without beginning or end), with constant amplitude, frequency, and phase offset. If for no other reason than its lack of a beginning and end, it is easy to see that "pure" sinewaves do not exist in nature—only in mathematics!

The *spectrum* of the "pure" sinewave described by Equation (9) can be represented thus:

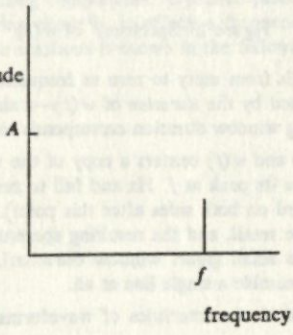


Figure 12: Spectrum[8] of a Sinewave with Amplitude A and Frequency f.

8. Actually, Figure 12 shows only the *positive-frequency half* of the complete frequency spectrum. Since the negative-frequency half of the spectrum of any sound waveform is just the left-right mirror image of the positive-frequency half it is

A sinewave that *has* a beginning and end at times t_1 and t_2 respectively may be represented by the mathematical expression

$$f_2(t) = A\sin(\omega t + \phi)\,,\ t_1 < t < t_2 \tag{10}$$

The spectrum of this waveform is *not* the same as the one shown in Figure 12, although as the duration between times t_1 and t_2 increases, the spectrum of the finite-duration sinewave approaches that of the infinite-duration sinewave.

In order to calculate the spectrum of the finite-duration sinewave described by Equation (10), we generally consider $f_2(t)$ to be the *product* of two infinite-duration waveforms, one being $f_1(t)$ and the other being

$$w(t) = \begin{cases} 1 & \text{if } t_1 < t < t_2 \\ 0 & \textit{otherwise} \end{cases} \tag{11}$$

Waveform $w(t)$ acts as a *rectangular window* function—the window is "open" between t_1 and t_2 and "closed" elsewhere. Therefore we have the relation

$$f_2(t) = f_1(t) \cdot w(t) \tag{12}$$

An important rule for calculating spectra is that if two waveforms are multiplied by each other, the spectrum of the resulting waveform is the *convolution* of the spectra of the two multiplied waveforms. Since the spectrum of $w(t)$ has the shape of the well-known *sinc* function:

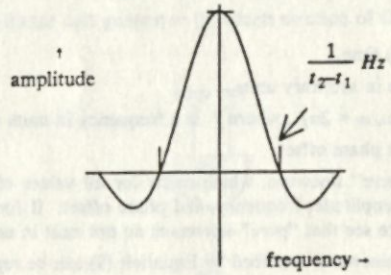


Figure 13: Spectrum[9] of $w(t)$

We see that the spectrum of $w(t)$ falls from unity to zero as frequency increases from zero to $1/(t_2-t_1)$ Hz. The rate of fall is therefore determined by the *duration* of $w(t)$—a short window duration corresponds to a broad spectral bandwidth, while a long window duration corresponds to a narrow spectral bandwidth.

Convolving the spectra of $f_1(t)$ and $w(t)$ centers a copy of the window spectrum at f Hz. As shown in Figure 13, the spectrum would have its peak at f Hz and fall to zero amplitude at $f \pm 1/(t_2-t_1)$ Hz (low-amplitude oscillations continue outward on both sides after this point). If t_2-t_1 is large (the duration of the window is long), then $1/(t_2-t_1)$ will be small, and the resulting spectrum would converge to a single vertical line at f Hz. Conversely, if t_2-t_1 is small (short window duration), then $1/(t_2-t_1)$ could be quite large, resulting in a spectrum that doesn't resemble a single line at all.

Spectral plots describe the physical characteristics of waveforms in the frequency domain. They are

usually not shown in spectral plots. Also, phase information is omitted from this representation of the spectrum. A *complete* representation of a spectrum requires *two* plots—one for amplitude and the other for phase.

9. In this case, both the negative- *and* positive-frequency sides of the spectrum are shown. As always, the negative-frequency side of the spectrum is the mirror image of the positive-frequency side.

really just another way to represent a waveshape in terms of the amplitudes and phases of the frequency components that comprise it. It is important to keep in mind that a spectral plot is not the same as a description of how a waveform will *sound* to a human being, for this rests on perceptual issues that a spectral plot simply does not address.

Nevertheless, spectral plots are an extremely useful tool for depicting the characteristics of waveforms in a way that is often closely related to how a waveform will be heard. Observation of spectra often will reveal whether a waveform will have a bright or dull timbre (i.e., tone quality), and how such timbral characteristics change over the duration of a sound. Spectral plots show us *what is going on in the sound itself*, so that we might gain insight into why a sound has the (perceived) timbre that it does. We may then manipulate sounds in terms of their spectra in order to gain particular timbral effects.

1.8.5. Basic Methods of Timbral Control

So far computer music researchers have identified four great categories of methods by which the timbre of a sound may be controlled. They are

1) *additive synthesis*, in which a number—possibly a *large* number—of sinusoidal components with time-varying amplitudes, frequencies, and phases are added together in order to control the distribution of energy at various frequencies across the spectrum,
2) *subtractive synthesis*, in which a spectrally rich waveform is filtered to remove unwanted energy at different frequencies across the spectrum,
3) *nonlinear synthesis*, in which certain mathematically-derived operations give rise to a variety of controllable distributions of energy across the spectrum, and
4) *physical modeling*, in which a mathematical model is used to simulate the physical (i.e., mechanical) properties of a vibrating system in order to generate the resulting patterns of vibration.

Each of these basic methods has been studied extensively both within and without the field of computer music, and a complete treatment of these topics would go far beyond this introduction to the use of the cmusic program. Each method has its benefits and liabilities, and they may be—and often are—combined in various ways for the generation of musical sounds. We will examine them only briefly here.

1.8.5.1. Additive Synthesis

Additive synthesis (sometimes inappropriately called *Fourier* synthesis) consists of adding together a wide variety of carefully controlled sinusoidal components to generate the final waveform. Each sinusoidal component has (in general) an independently time-varying amplitude, frequency, and phase.

For sound synthesis, phase is typically discounted because of the relative insensitivity of the ear to differing *constant* phase relations among components. Dynamic phase relations often can be handled through frequency control since a time-varying phase is, in effect, a frequency. A cmusic instrument for generating an individual component for additive synthesis is shown in the following diagram.

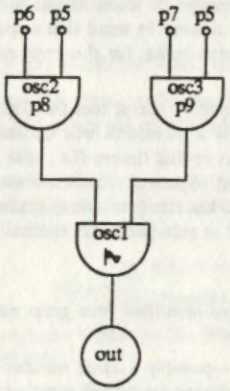


Figure 14: cmusic Instrument for a Single Additive Synthesis Component

In this instrument the wavetable used by *osc2* will control the time-varying amplitude of the component while the wavetable used by *osc3* will control its time-varying frequency. Instead of a wavetable reference, the instrument shown in Figure 14 indicates how note parameters may be used to specify which wavetable is to be used for a particular note. If $p8$ has the value 2.0 (actually, if $2 \le p8 < 3$), *osc*2 will use $f2$ as its wavetable. $p9$ similarly selects a frequency-control wavetable for *osc*3. Since cmusic allows any number of notes to be played simultaneously on a single instrument, additive synthesis may be achieved by playing multiple notes simultaneously on this simple instrument, each referring to different wavetables for frequency and amplitude control. The details of the wavetables themselves then control the behaviors of the individual sinusoidal components.

One can see at a glance that the number of required wavetables can become quite large if they are used in this way. Furthermore, different wavetables are likely to be required for each note! This is the basic difficulty involved with additive synthesis: the amount of control data tends to be huge. Nevertheless, additive synthesis is the most general method available for controlling timbre[10], and cmusic provides several alternatives for achieving component control.

There are several *gen* functions that can be used to *gen*-erate control functions for additive synthesis. We've already seen examples of of *gen* statements that fill wavetables with sine and envelope control functions. There are probably as many ways to generate wavetable data as there are wavetables to be generated—cmusic provides a variety of them that behave in various useful ways.

gen functions are free-standing programs that can be run independently of cmusic. This allows users to write their own *gen* programs if cmusic doesn't already provide a convenient way to specify your favorite waveshape. Many of the cmusic *gen* programs have names like "gen1", "gen2", etc., for historical reasons, but some have more descriptive names like "chubby", which generates sums of Chebychev polynomials for use in certain types of nonlinear synthesis. These "official" *gen* functions are all described later in this paper.

One important issue raised by this example is the need for the cmusic user to keep track of the total amplitude of the output waveform—it cannot exceed ±1.0, even for a single sample, without incurring *clipping*. cmusic "clips" any output samples that exceed this range (i.e., if an output sample is greater than 1.0, it is replaced with 1.0, and if it is less than –1.0, it is replaced with –1.0) because the DAC system simply cannot produce a greater amplitude in the output waveform. When many note events occur simultaneously—as they might for additive synthesis—it is important to check that the total amplitude does not exceed legal

10. For further information on additive synthesis, the interested reader is referred to "Signal Processing Aspects of Computer Music" by James A. Moorer, in *Digital Audio Signal Processing: An Anthology*, William Kaufmann, Inc., 1985.

limits. Therefore we might choose a maximum amplitude of $1/N$ for individual components of an N-component waveform and scale the amplitudes of individual components relative to this maximum value.

1.8.5.2. Subtractive Synthesis

Additive synthesis builds up a complex waveform out of elemental sinusoidal building blocks. The relationship among the components may be harmonic (all integer multiples of a single fundamental frequency) or inharmonic, as desired. Subtractive synthesis *starts* with a complex waveform and removes unwanted spectral energy by lowering their amplitudes with some type of *filter*. The complex waveform for subtractive synthesis is typically either a broadband harmonic waveform or some type of broadband noise.

Like their analog counterparts, digital filters have signal inputs, signal outputs, and one or more control inputs that determine their operation. With appropriate choices of complex waveforms and filtering parameters, a wide variety of sounds can be synthesized. Most notably, subtractive synthesis is particularly effective for generating the sounds of human *speech*[11].

In order for achieve timbral variation with subtractive synthesis, it is generally necessary for the filter to be time-varying, i.e., its parameters—hence its operation—usually change over the duration of a note event. Digital filtering is an enormously well-developed subject, which means that there is much to know about it. Nevertheless, all possible digital filters can be described by a single, relatively straightforward relation:

$$y(n) = \sum_{i=0}^{M} a_i x(n-i) - \sum_{j=1}^{N} b_j y(n-j) \qquad (13)$$

where:

$y(n)$ is the output signal,

$x(n)$ is the input signal,

M is the number of *poles*,

N is the number of *zeros*,

a_i is a set of M *feedback coefficients*, and

b_j is a set of N *feedforward coefficients*.

The *poles* of a filter determine the behavior of its *resonances*—frequency regions that the filter tends to emphasize, and the *zeros* of a filter determine the behavior of its *antiresonances*—frequency regions that the filter tends to de-emphasize. The center frequencies and bandwidths of the poles and zeros of a digital filter are determined by the filter coefficients. Practically the entire problem of digital filtering boils down to finding ways to obtain these coefficients for specific filtering tasks[12]. In a way, control of these coefficients in subtractive synthesis corresponds to control of the component amplitudes and frequencies in additive synthesis.

cmusic provides two basic filter unit generators that may be combined in various ways. Their operation is very similar, but they differ in the manner in which the filter operation is specified.

The basic cmusic filter is a unit generator called *flt*, which implements the so-called *general second-order filter stage* described by the relation:

$$y(n) = g[a_0 x(n) + a_1 x(n-1) + a_2 x(n-2) + b_1 y(n-1) + b_2 y(n-2)] \qquad (14)$$

This equation is the same as the previous one with both M and N set to 2 and with an extra overall gain factor g. By controlling the a and b coefficients, the cmusic user can use *flt* as a time varying lowpass, highpass, bandpass, or band-reject filter.

For example, a simple two-pole resonator can be specified by setting the *flt* coefficients as follows:

11. For further information on speech synthesis the interested reader is referred to *Digital Processing of Speech Signals* by L. R. Rabiner and R. W. Schafer, Prentice-Hall, Inc., 1978.

12. For further information on digital filtering the interested reader is referred to "An Introduction to Digital Filter Theory" by J. O. Smith, in *Digital Audio Signal Processing: An Anthology*, William Kaufmann, Inc., 1985.

$$y(n) = Gx(n) + (2r\cos\theta)y(n-1) - r^2y(n-2) \tag{15}$$

where:

r is the *pole radius*, which determines the bandwidth (or Q) of the resonance,

θ is the *pole angle*, which determines the center frequency of the resonance, and

G is an arbitrary gain factor.

The pole radius of the resonance is related to the frequency bandwidth according to the following approximation:

$$r \approx e^{-\pi B/R} \tag{16}$$

where B is the desired bandwidth in Hz (the width of the resonant region once it has fallen from the peak by a factor of 3 dB), and R is the sampling rate. The pole angle is proportional to

$$\theta = 2\pi\frac{f}{R} \tag{17}$$

where f is the desired center frequency in Hz. The *flt* parameters a_1 and a_2 would be set to 0.0 in this case (thereby "turning off" the antiresonances), and both g and a_0 would typically be set to 1.0 (they might have to be adjusted to prevent output clipping, depending on the signal being filtered).

The amplitude of the output signal depends on both the action of the filter and the nature of the input signal, making it sometimes difficult to predict. For this reason, cmusic provides another filter called *nres*[13] which has a *normalized gain* of 1.0 at its peak resonance point. *nres* implements the filter described by the equation

$$y(n) = G[x(n) - rx(n-2)] + (2r\cos\theta)y(n-1) - r^2y(n-2) \tag{18}$$

with G set to $1-r$.

The predictable gain behavior of *nres* makes it easier to use in many cases than the more general *flt* unit generator. Furthermore, the parameters of *nres* are specified directly in terms of center frequency and bandwidth (rather than in terms of filter coefficients), again increasing its convenience.

A typical use of *nres* might start with a harmonic-rich waveform such as that produced by the *band-limited pulse* unit generator *blp*. *blp* uses a closed-form summation formula to generate an arbitrary (finite) number of equal-strength harmonic components with arbitrary spacing along the frequency axis. Band-limited pulses are useful in digital music synthesis because they provide a source of harmonic-rich waveforms that do not produce *foldover* (*aliasing*) when properly used. *blp* operates by evaluating the relation

$$y(n) = \sum_{k=0}^{N-1} \sin(\alpha + k\beta) = \sin\left[\alpha + \frac{(N-1)\beta}{2}\right]\sin\left(\frac{N\beta}{2}\right)\csc\left(\frac{\beta}{2}\right) \tag{19}$$

where $y(n)$ is the output of *blp*, $\alpha = 2\pi nf_1/R$ and $\beta = 2\pi nf_2/R$.

We can choose f_1, f_2, and N to generate a spectrum which has the following form:

13. *nres* is based on the filter described in "A Constant-Gain Digital Resonator Tuned by a Single Coefficient" by Julius O. Smith and James B. Angell, *Computer Music Journal*, volume 6, number 4, Winter, 1982.

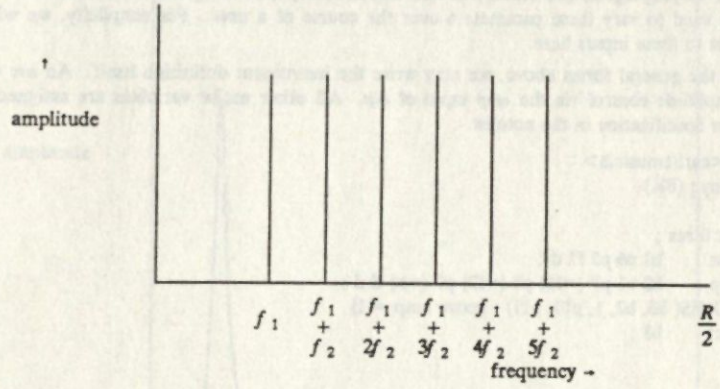


Figure 15: Spectrum of a *blp* Output Waveform for N =6.

blp allows us to control the placement of exactly N frequency components starting at f_1 Hz with a spacing of f_2 Hz. Appropriate choices therefore can ensure that no component exceeds the Nyquist frequency (one-half the sampling rate).

In order to build a cmusic instrument that generates a filtered band-limited pulse waveform, we need first to look up the general form of the unit generator statements for *blp* and *nres*, which are as follows:

```
blp output[b] amp[bvpn] incr1[bvpn] incr2[bvpn] n[bvpn] sum1[dpv] sum2[dpv] ;

nres    output[b] input[bvpn] gain[bvpn] cf[bvpn] bw[bvpn]
          t1[dpv] t2[dpv] t3[dpv] t4[dpv] t5[dpv] t6[dpv] t7[dpv] t8[dpv] t9[dpv] t10[dpv] ;
```

Each unit generator parameter is described with an indication of its purpose and a bracketed list of the types of cmusic parameters that may be used to control it. For instance we see that the first parameter for the *blp* unit generator is its signal output (*out*), and that it must be connected to an i/o block (*b*). The second *blp* parameter is an amplitude control input, and it may be an i/o block (*b*), a global variable[14] (*v*), a note parameter (*p*), or a constant number or expression (*n*). The third *blp* parameter is an *increment* input, indicating that it controls a frequency, and that the "Hz" postoperator should be used to scale it in the manner of an oscillator increment value. *incr* 1 and *incr* 2 control f_1 and f_2, respectively. The last two *blp* parameters are internal state variables that normally will be assigned to dynamic variables (*d*).

The *nres* unit generator has a signal output, a signal input, and controls for gain, center frequency, and bandwidth. Ten state variables (all normally of type *d*) are also required. Later in the documentation for *nres*, we discover a *macro* (shorthand) definition that obviates the need to carefully write 10 *d*'s at the end of the *nres* statement:

```
#define NRES(out,in,gain,cf,bw) nres out in gain cf bw d d d d d d d d d d
```

We may avail ourselves of this shorthand merely by placing the statement

```
#include <carl/cmusic.h>
```

at the head of the cmusic score, which is a highly recommended practice for all scores.

We also can see from the general statement *nres* statement that the center frequency and bandwidth

14. cmusic global variables will be explained later on.

ts may be time-varying signals (i/o blocks) as well as constants, indicating that oscillators and other unit erators may be used to vary these parameters over the course of a note. For simplicity, we will supply constant values to these inputs here.

Referring to the general forms above, we may write the instrument definition itself. An *osc* unit generator provides amplitude control via the *amp* input of *blp*. All other major variables are assigned to note meters for later specification in the notelist.

```
#include <carl/cmusic.h>
#define Fnyq (8K)

instrument 0 res ;
        osc     b1 p6 p5 f1 d ;
        blp     b2 b1 p7 {=f1} p8 {=f2} p9 {=n} d d ;
        NRES( b3, b2, 1, p10, p11) ; {extra amp = 1}
        out     b3 ;
end ;

ENV(f1) ;
```

Example 3: cmusic Instrument for Resonated *blp* Waveform

then can play this instrument with a *note* statement such as:

p1]	{p2}	{p3}	{p4}	{p5}	{p6}	{p7}	{p8}	{p9}	{p10}	{p11}
	{time}	{ins}	{dur}	{1/dur}	{amp}	{f1}	{f2}	{n}	{cf}	{bw}
ote	0	res	3	p4sec	-18dB	220Hz	220Hz	Fnyq/220	1000Hz	200Hz ;

Example 4: Sample Note Statement for Resonated *blp* Waveform

We see in the instrument definition above our first example of a *macro definition*. The shorthand yq" is defined to stand for "(8K)", which is a cmusic expression meaning 8·1024 = 8192. Judging by the thand chosen, we infer that this value must be equal to one-half the sampling rate. The shorthand is l in the expression for note parameter *p*9 to calculate the maximum possible value for *n* (the number of uency components *blp* is to generate—the calculated non-integer value is *truncated* by *blp*). cmusic macro nitions work exactly as they do in the C programming language. In fact, cmusic passes the score through ctly the same preprocessor that the C compiler uses, allowing the user to define macros (with or without uments), include special files, conditionally execute parts of a score, and so on.

The *NRES* statement in the instrument definition shows how a macro with arguments can be used. notation is positional, just as in a cmusic note statement. The first item inside the parentheses informs *nres* unit generator about what output to use, the second item specifies the input, and so on.

The note statement itself also demonstrates some new features. Note parameter *p*5 is set to "p4sec"—"sec" postoperator operates in such a way that "ξsec" is equal to "1/ξHz", as required by the amplitude elope control oscillator. The peak amplitude is given as "-18dB" which evaluates to approximately 0.125, e every 6 dB corresponds to an amplitude factor of about 2, and the "0dB" reference value is 1.0. f_1 and for the *blp* unit generator are both set to "220Hz" (note that *blp* requires increment-type inputs here), ch specifies a harmonic spectrum based on *A220*. Since 8192/220 is equal to 37.236 · · · , *blp* will generate aveform with 37 equal-strength components.

The *blp* waveform is filtered via an *nres* unit generator with a center frequency of 1000 Hz and a dwidth of 200 Hz giving it a frequency response curve as shown in the following diagram.

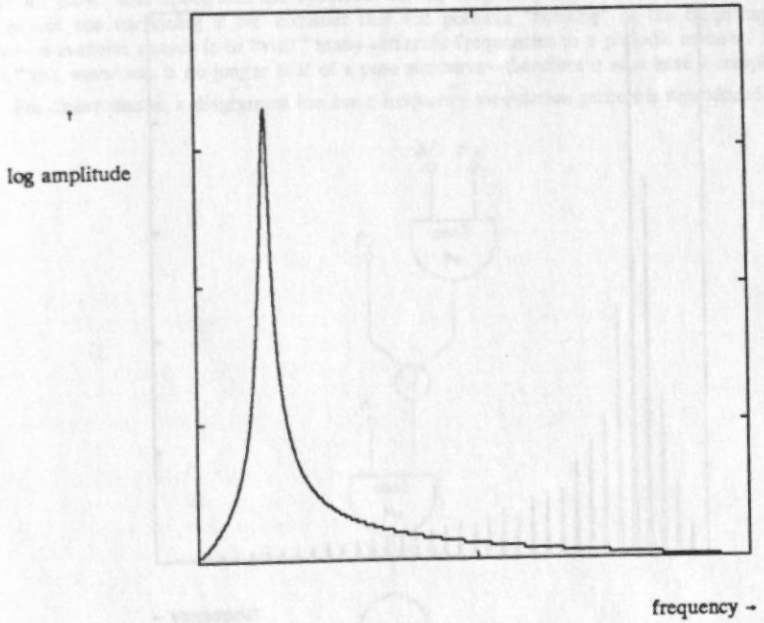


Figure 16: Frequency Response of the *nres* Unit Generator
as Used in Example 4.

In effect, the curve in Figure 16 depicts the frequency-dependent gain of the *nres* filter as we have used in Example 4. The horizontal axis in Figure 16 goes from zero to half the sampling rate (a 16,384 = 16K Hz sampling rate is assumed). The vertical axis shows amplitude on an arbitrary logarithmic (dB) scale. We can see that the peak in the response is centered at 1000 Hz. The frequency band under the curve is approximately 200 Hz wide when the curve is 3 dB lower than its peak value on either side. We can also see from this curve that *nres* has two zeros (*antiresonances*) at 0 Hz and at the Nyquist rate.

Like all filters, *nres* is basically a frequency-dependent amplifier/attenuator. After being processed by *nres*, the amplitude of the individual spectral components of the *blp* output will be determined by where they happen to fall under the *nres* frequency response curve. In effect, *nres* superimposes this curve on the "flat" spectrum of the waveform generated by *blp*. The spectrum of the output waveform produced by instrument *res* as used in Example 4 therefore has a shape as shown in the following diagram.

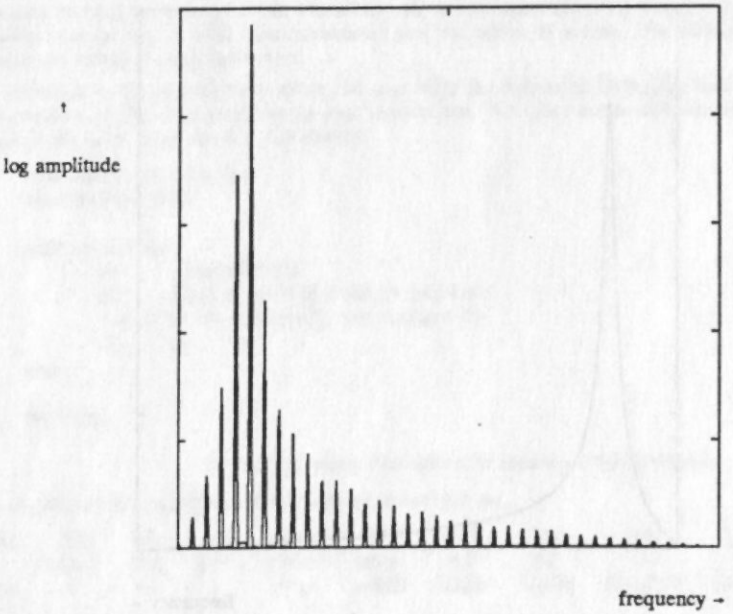


Figure 17: Spectrum of the Output Waveform of Instrument *res* as Used in Example 4.

We see from Figure 17 that the fundamental frequency component (at 220 Hz) is considerably weaker than the second, third, fourth, and fifth harmonics. Above the fifth harmonic, however, the partials die away rapidly with increasing frequency (though many are present at significant amplitudes).

1.8.5.3. Nonlinear Synthesis

So far we have considered mostly *linear* methods of waveform synthesis. In this context, "linear" means that the only frequency components present in the output waveform are those that are placed there directly. If frequencies are present in the output of a synthesis process that are not present at its input, that process is said to be *nonlinear*.

The term "nonlinear," however, is not a very good description of how such processes work. Describing a process as nonlinear only says what that process is not—not what it is!

Actually, the synthesis process embodied in the *blp* unit generator described above is nonlinear, since we feed it only two frequencies and another number but many other frequencies are generated as a result! Another nonlinear process we have already considered is vibrato synthesis. The spectrum of a vibrato waveform actually contains many more frequencies than just those of the main pitch and the vibrato waveform.

Vibrato synthesis is just a special case of a more general process called *frequency modulation*, or simply, *FM*[15]. Just as we have already seen that the spectrum of a sinewave undergoing amplitude modulation is no

15. In fact, it was due to some curious results obtained while experimenting with extreme vibrato that John Chowning discovered that the well-known methods of frequency modulation could be used to synthesize an astonishing variety of different timbres. The interested reader is referred to "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation" by John Chowning, in *Foundations of Computer Music*, MIT Press, 1985 (this article appeared earlier in both the

longer a "pure" sine spectrum, the spectrum of any frequency modulated waveform is similarly complex. This is not too surprising if we consider that the periodic "bending" of the frequency of the main—or *carrier*—waveform causes it to "visit" many different frequencies in a periodic manner. Because it is being "bent," the waveform is no longer that of a pure sinewave—therefore it *must* have a complex spectrum.

For convenience, a diagram of the basic frequency modulation process is reproduced below.

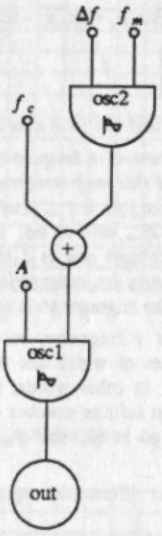


Figure 18: The Frequency Modulation Process

The continuous-time output waveform associated with the frequency modulation process shown in Figure 18 may be represented mathematically as follows:

$$f(t) = A\sin[\omega_c t + I\sin(\omega_m t)] \tag{20}$$

where:

- A is an arbitrary amplitude,
- ω_c is the *carrier* frequency (in radian units, i.e., $\omega_c = 2\pi f_c$ where f_c is the carrier frequency in Hz),
- ω_m is the *modulating* frequency ($= 2\pi f_m$), and
- I is called the *modulation index*.

In frequency modulation terminology, the *modulation index* is defined as the ratio of the *frequency deviation* to the *modulating frequency*:

$$I = \frac{\Delta\omega}{\omega_m} = \frac{\Delta f}{f_m} \tag{21}$$

where Δf is the *frequency deviation* (more or less equivalent to vibrato depth—see Figure 8).

We can obtain insight into the spectrum of a frequency modulated waveform by examining the *trigonometric expansion* of Equation (20):

Computer Music Journal [1977] and *Journal of the Audio Engineering Society* [1973]).

$$
\begin{aligned}
f(t) = A\sin[\omega_c t + I\sin(\omega_m t)] = A\Big\{ & J_0(I)\sin\omega_c t \\
& + J_1(I)[\sin(\omega_c+\omega_m)t - \sin(\omega_c-\omega_m)t] \\
& + J_2(I)[\sin(\omega_c+2\omega_m)t + \sin(\omega_c-2\omega_m)t] \\
& + J_3(I)[\sin(\omega_c+3\omega_m)t - \sin(\omega_c-3\omega_m)t] \\
& + \cdots \Big\}
\end{aligned}
\tag{22}
$$

where $J_n(I)$ is the *Bessel function of the* n^{th} *order* and the *first kind*.

We see from Equation (22) that the spectrum of a frequency modulated waveform consists of a component at the carrier frequency f_c, plus a pair of *sideband* components centered about the carrier frequency at a distance equal to the modulating frequency, i.e., $f_c \pm f_m$, plus another sideband pair at twice the modulation frequency distance from the carrier, $f_c \pm 2f_m$, and so on. Furthermore, the amplitude of the carrier is shown as $J_0(I)$, the amplitude of the first sideband pair is $J_1(I)$, the second sideband pair amplitude is $J_2(I)$, and so on. Finally, some of the sideband pairs are symmetrical and others are antisymmetrical, as indicated by the alternating plus and minus signs in the trigonometric expansion.

Equation (22) shows that the spectrum of a frequency modulated waveform consists of an *infinite* number of frequency components, the amplitudes of which are related to an infinite succession of Bessel functions evaluated at the modulation index, I. In other words, the simple process of modulating the frequency of one sinewave with another produces an infinite number of spectral components! This is the signature of the nonlinear process—two frequencies go in (f_c and f_m), and an infinite number come out (f_c, $f_c \pm f_m$, $f_c \pm 2f_m$, ...).

Bessel functions are solutions to a particular differential equation known as Bessel's Differential Equation:

$$x^2\ddot{y} + x\dot{y} + (x^2-n^2)y = 0 \quad n \geq 0 \tag{23}$$

which has the so-called Bessel functions of the first kind as solutions:

$$
\begin{aligned}
J_n(x) &= \frac{x^n}{2^n\Gamma(n+1)}\left\{1 - \frac{x^2}{2(2n+2)} + \frac{x^4}{2\cdot4(2n+2)(2n+4)} - \cdots\right. \\
&= \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{n+2k}}{k!\,\Gamma(n+k+1)}
\end{aligned}
\tag{24}
$$

While a full treatment of Bessel functions goes beyond the scope of this introduction, we can gain sufficient insight into them by observing their graphs all plotted to the same scale.

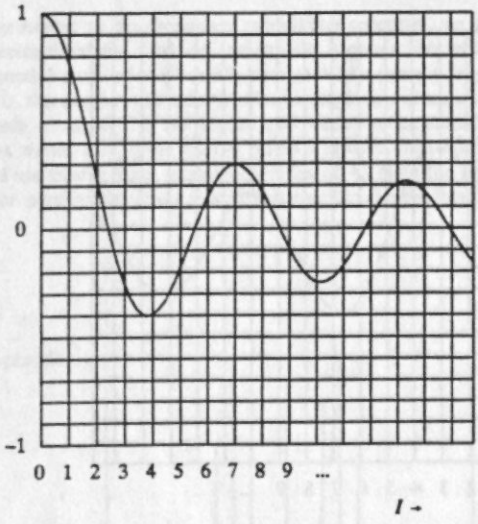


Figure 19: Bessel Function J_0 as a Function of I

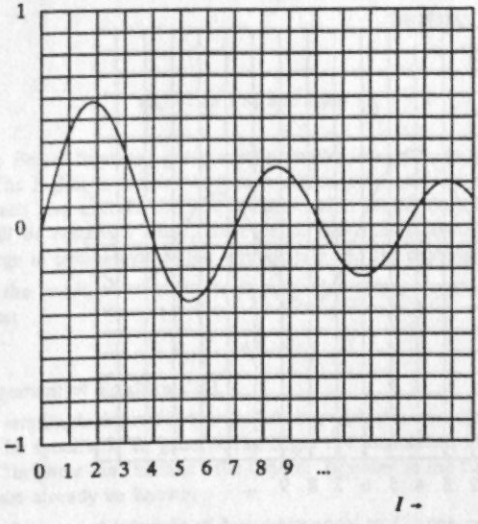


Figure 20: Bessel Function J_1 as a Function of I

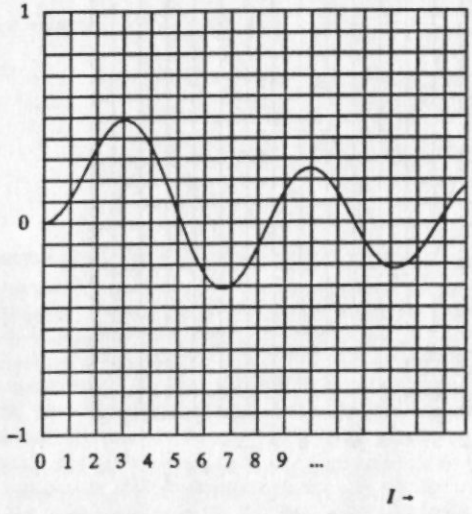


Figure 21: Bessel Function J_2 as a Function of I

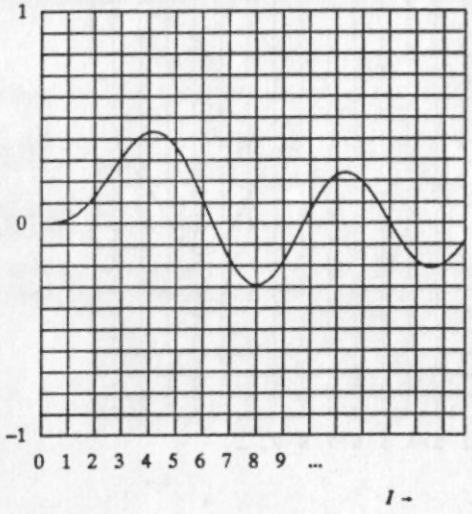


Figure 22: Bessel Function J_3 as a Function of I

The zero-th order Bessel function, $J_0(I)$ starts at 1.0 and traverses a more or less "damped cosinusoidal" shape. All other orders start at 0.0 and traverse a more or less "damped sinusoidal" shape. Each order is a bit smaller than the previous one, and each one "dies away" as the argument I increases.

Relating this information to the frequency modulation spectrum, we see that as the modulation index increases, the overall strength of the sideband components becomes less and less, but there will be more and more of them! If the modulation index (I) is 0.0, energy will be present only at the carrier frequency. For a modulation index of 1.0, the carrier will have a relative amplitude of about 0.85, the first sideband pair will have a relative amplitude of about 0.4, the second pair will have an amplitude of about 0.1, and so on. A larger modulation index would spread the energy across a greater bandwidth. Figure 23, for example, shows the spectrum of an FM waveform for a modulation index of 4. Note the symmetry of the sidebands as they spread outward from the carrier frequency at intervals equal to the modulating frequency.

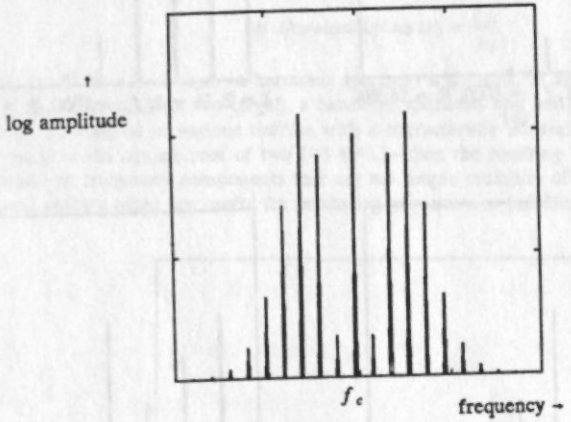


Figure 23: FM Spectrum for I = 4.

The fact that the Bessel functions get generally smaller in amplitude with increasing order is important for digital synthesis. The higher-order sideband pairs extend outward from the carrier frequency—eventually some of them may reach and exceed the Nyquist limit, after which they fold over. The amplitude of the aliased components will be relatively small, however, so that this foldover is not usually audible unless the carrier frequency is large in comparison to the Nyquist rate and the modulation index is also large.

In order to set the modulation index to a particular value, we must choose a frequency deviation according to the relation

$$\Delta f = I \cdot f_m \tag{25}$$

which is just a rearrangement of Equation (21).

By choosing the amplitude input for the modulating oscillator according to Equation (25), any desired modulation index may be specified. In general, the larger the modulation index, the larger the bandwidth of the spectrum, and the "brighter" or "shriller" the timbre. In order to use Equation (25), however, the modulating frequency f_m must already be known.

Since the sidebands occur at intervals of frequency equal to f_m, one possible choice would be to set f_m to the carrier frequency, f_c. This would yield spectral components at f_c, $f_c \pm f_c$, $f_c \pm 2f_c$, and so on. The first sideband pair would therefore occur at frequencies 0 and $2f_c$ Hz, the second pair would have the frequencies $-f_c$ and $+3f_c$ Hz, and so on, yielding a harmonic spectrum with a fundamental frequency of f_c Hz.

Recalling that $\sin(-x) = -\sin(x)$, we can conclude that the *negative frequencies* produced by the FM process would just "wrap around" to positive frequencies, undergoing a 180° phase shift along the way, where they combine in amplitude with any spectral energy already present at those frequencies. Despite the 180°

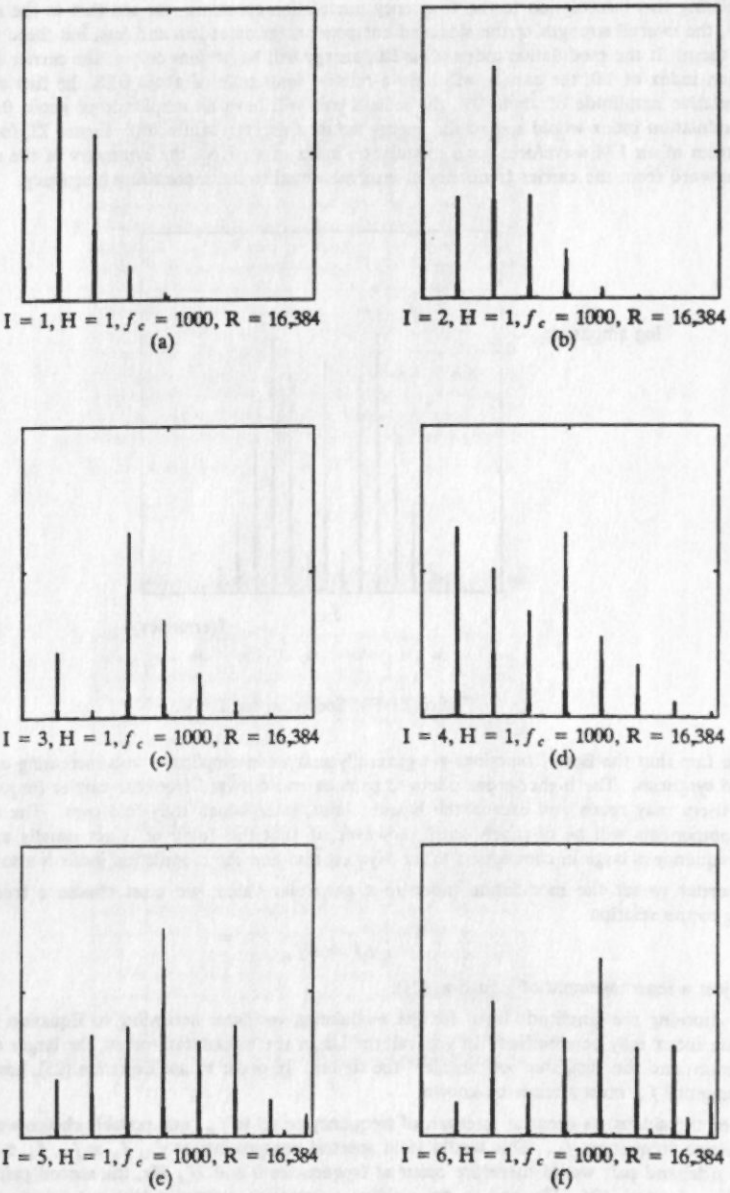


Figure 24: Spectra of FM Waveforms for Modulation Indices 1 through 6.
$H = f_c/f_m = 1$ in all cases. Note foldover in examples (d), (e) and (f).

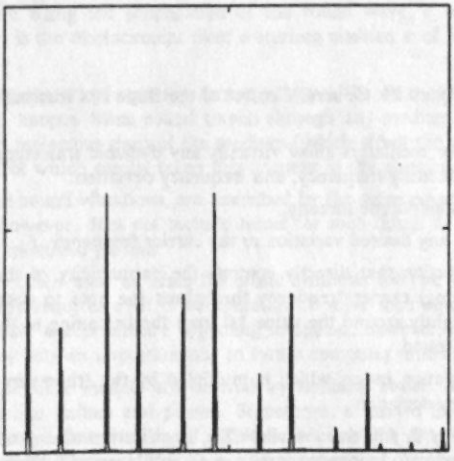
phase shift, complete cancellation is unlikely because the negative-frequency components will generally have different amplitudes than the positive frequency components with which they collide. Due to the antisymmetry of some of the spectral components, this 180° phase shift may also negate an already negative amplitude, resulting in constructive rather than destructive interference.

Many other choices are possible for f_m. By setting $f_m = 2f_c$, we again obtain a harmonic spectrum, but this time the even-numbered harmonics of f_c will be absent. In fact, *any* integer multiple of the carrier frequency used as a modulating frequency will result in a harmonic spectrum.

Aficionados of FM synthesis like to define a quantity H called the *harmonicity ratio* of an FM waveform:

$$H \text{ (harmonicity ratio)} = \frac{f_m}{f_c} \tag{26}$$

We have seen that if H is an integer, a harmonic spectrum will result. It also turns out that if H is equal to $1/N$, where N is an integer (not too large!), a harmonic spectrum also results. Such harmonic spectra often are useful in the production of various timbres with a characteristic musical pitch (see Figure 24). But if H is irrational—such as the square root of two (≈1.414...)—then the resulting FM spectrum will be *inharmonic*, i.e., it will consist of frequency components that are *not* simple multiples of a single fundamental frequency. Such inharmonic spectra often are useful for producing percussive or bell-like timbres (see Figure 25).



I = 3, H = 1.414, f_c = 1000, R = 16,384

Figure 25: Spectrum of an Inharmonic FM Waveform

Different timbres are produced by the frequency modulation process according to how the modulation index and harmonicity are made to vary throughout a sound event. Of course, amplitude and frequency variation may be applied as well.

A simple but quite general control section for the basic instrument depicted in Figure 18 is shown in Figure 26.

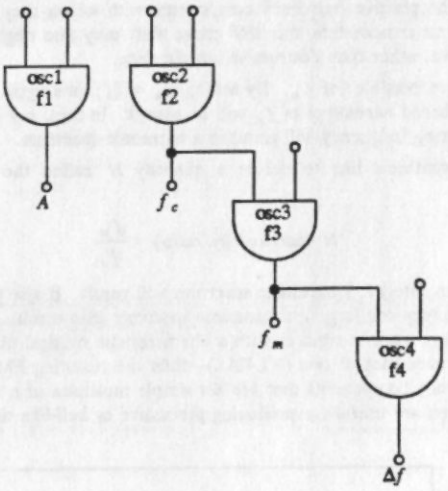


Figure 26: General Control of the Basic FM Instrument

Wavetables for the four oscillators allow virtually any dynamic trajectory to be imposed in the amplitude, carrier frequency, modulating frequency, and frequency deviation.

Wavetable f 1 controls amplitude directly.

Wavetable f 2 imparts any desired variation to the carrier frequency, f_c.

Wavetable f 3 is a function that directly controls the harmonicity of the FM waveform—it is simply multiplied by the (time-varying) carrier frequency throughout the note to obtain the modulating frequency. For example, if f 3 varies slightly around the value 1.0 from the beginning to the end of a note, so would the harmonicity of the resulting sound.

f 4 specifies the modulation index, which is multipled by the (time-varying) modulating frequency to obtain the requisite frequency deviation.

Suitable choices of f 1, f 2, f 3, and f 4 allow the specification of *any* of the musical timbres that the basic FM instrument can produce. Increment inputs to all oscillators will typically be $1/p4$ Hz, though other values may be used to obtain such things as periodic undulation of harmonicity.

1.8.5.4. Physical Modeling

The final synthesis method considered here goes under the somewhat vague appellation of *physical modeling*. What is meant by a *physical model* in this context is a mathematical formulation of how a given physical system will vibrate. Insofar as the mathematics truly captures the relevant properties of the physical system, solving the equations should result in the vibration pattern associated with that system. Since vibration is a form of motion, the necessary mathematics almost always involves that particular kind of mathematics invented to describe motion: *calculus*.

Calculus is a wonderful subject, and a really important achievement in mathematics[16], but before the

16. Albert Einstein was once asked what he thought was the greatest single intellectual achievement of human beings. He is reported to have responded, without hesitation, that Newton's development of the calculus in order to describe motion was the greatest such achievement in history. Today we have a little more information. The mathematician Leibnitz, working independently, formulated the rules of calculus at about the same time as Newton. However, there is now additional evidence that calculus may have been invented even earlier in China.

reader gets too worried, rest assured that we won't go into such mathematical methods here.

On a qualitative level, however, a physical model is derived from *reasoning* about the physical characteristics of a vibrating system. For sound waves, the basic features of air (a gas) form the basis for understanding how sound works. Such basic features are as follows. Assume that *here* is a little region of space, and *there* is any neighboring little region.

1. When air moves, the density of air particles changes, since moving a little air from *here* to *there* decreases the amount of air *here*, and increases the amount of air *there*.
2. Changes in air density result in changes in air pressure, since there's more air squeezed into *there* than *here* after it has moved.
3. Having more air *there* than *here* makes it "want" to move, since the pressure is no longer equal everywhere.

These are very simple ideas, and they all can be described precisely in a mathematical fashion. Putting together the mathematical descriptions of these three features of air leads directly to what physicists call the *wave equation*, which describes the behavior of sound in all matter, not just air! One form of the wave equation is as follows:

$$\frac{\partial^2 \chi}{\partial x^2} = \frac{1}{c^2} \frac{\partial^2 \chi}{\partial t^2} \tag{27}$$

where t is time, x is position along the propagation of the sound wave, c is the speed of sound in the medium (like air!), and $\chi(x,t)$ is the displacement from a starting position x of a portion of air (or whatever) at time t.

Whether you understand the wave equation or not, it should be clear that this mathematical relationship states precisely what will happen when sound travels through any medium. It will be affected by such things as the temperature and molecular mass of the medium (which affect the speed of propagation, c), and many other things as well, all of which can be taken into account in the physical model, if desired.

Vibrating strings, like all sound vibrations, are described by the wave equation. The simple form of the wave equation given above, however, does not include terms for such things as stiffness in the string, which causes strings to have slightly mistuned partials.

Physical models are not often used as bases for music synthesis for two reasons: they are mathematically complicated and it usually requires a lot of computation to solve such equations by numerical methods available on computers. Notable exceptions are beginning to appear, however, on both of these fronts, hinting that physical modeling may play an important role in future computer music synthesis techniques.

As mentioned above, physical models are derived by thinking about the physical (i.e., mechanical) behavior of vibrating systems like guitars and pianos. Sometimes, a shrewd line of reasoning will lead to a rather simple mathematical model that works well. This is precisely the type of reasoning that resulted in the algorithm devised by Kevin Karplus and Alex Strong for synthesizing plucked string and drum timbres[17].

"Working well" means both that the model leads to a computer program that runs with reasonable speed and that it sounds good. Most physical models have never really been explored sonically because they require huge amounts of particular types of calculations. Special-purpose Very Large Scale Integrated (VLSI) circuits are now being developed that may perform these particular types of calculations at blinding speeds. If so, physical modeling may become a common—as well as an important—method of music synthesis.

17. The Karplus-Strong algorithm (as extended by David Jaffe, Julius Smith, and Mark Dolson) forms the basis for the cmusic *fltdelay* unit generator. The interested reader is referred to "Digital Synthesis of Plucked-String and Drum Timbres" by Kevin Karplus and Alex Strong, *Computer Music Journal*, volume 7, number 2, Summer, 1983.

2. cmusic Reference Manual

2.1. General

cmusic translates a *score* into a *digital signal*. The score is ASCII text that has been prepared manually by the cmusic user with a text editing program, or it may have resulted from a computer program. By convention, cmusic score files are named with a ".sc" extension (cmusic does not enforce this, however).

The digital signal is a sample stream suitable for digital-to-analog conversion once it is complete. Properties of the digital signal are controlled by *instruments* that are defined in the score, and by note events "played" on these instruments.

A cmusic score consists of *score statements*. Each score statement contains one or more *fields* that are separated by *whitespace* (blanks, tabs, or newlines) and/or commas. Each score statement is terminated with a semicolon (;).

The first field of a cmusic score statement always specifies a *command*. The format of the remainder of the statement depends on the command given. Individual fields must not contain embedded whitespace.

Statements are usually typed one per line, but several semicolon-delimited statements may be placed on one line, and one statement may continue over more than one line.

cmusic passes the score through the C language *preprocessor* before it interprets the score. The preprocessor allows specification of *include files* and *macros*—both with and without *arguments*. Macros allow user-defined shorthand notations for common or complex score statement idioms. A set of standard macro definitions is provided for the most commonly used idioms.

Comments may be inserted into the score between curly braces ({}). Text between curly braces is ignored completely by cmusic[18]. Comments may be nested to any depth (i.e., comments within comments are allowed).

18. However, in the current implementation of cmusic curly brace-enclosed comments are *not* ignored by the preprocessor.

The following table lists the currently available cmusic commands. Since cmusic is a research tool, the set of available commands occasionally changes.

cmusic Commands

Command	*Short Form*	*Meaning*	*Description*
set	set	Set global parameter	Specifies global parameters for a cmusic run such as the sampling rate, default wavetable length, number of audio channels in the output signal, etc.
instrument	ins	Begin instrument definition	Specifies that the following statements describe an interconnection of cmusic unit generators in a configuration designed to produce a particular type of sound.
end	end	End instrument definition	Terminates an instrument definition.
generate	gen	Generate wavetable	Specifies that a wavetable is to be filled with values computed by a specified program, and passes any needed control information to that program.
note (play)	not (pla)	Play a note	Specifies that a note event is to be "played" on one of the defined instruments, and passes any needed control information to that instrument.
section	sec	Terminate a score section	Specifies the end of a series of note events, resets the action time to zero, and continues score processing.
variable	var	Define global variable value(s)	Specifies the value(s) to which one or more global variables is (are) to be set (cmusic global variables may have either numerical or string values).
merge	merge	Begin a merge section	A merge section of note events is sorted into nondecreasing time order, merged with other (optional) sorted merge sections, and processed.
endsec	endsec	End merge section	Specifies the end of a merge section after a merge command has been used.
endmerge	endmerge	End of a group of merge sections	Specifies that the end of a group of merge sections has been reached.
terminate	ter	Terminate score	Specifies that the end of the score has been reached.

cmusic scores may be organized in a wide variety of ways. The following general outline illustrates a nominal organization that may help a cmusic score to succeed.

General Outline of a cmusic Score

Part	*Operations*	*Commands*
I.	OPTIONS	set
II.	INSTRUMENT DEFINITION(S)	ins unit generator statements end
III.	WAVETABLE GENERATION	gen
IV.	NOTELIST	note (play) sec var merge endsec endmerge
V.	TERMINATION	ter

2.2. Expressions

Most fields in cmusic score statements contain numerical values (they need not be *typed* as numerical values if macro definitions are used). An *expression* may be typed in any field that requires a constant numerical value.

cmusic interprets the expression by executing the specified calculation—the resulting number becomes the value of the field containing the expression. For example, we may type either "4" or "2+2" to specify the same thing. Remember that embedded spaces are forbidden within fields. The following table summarizes the possibilities for cmusic expressions.

cmusic Expressions		
OPERANDS	**DESCRIPTION**	**EXAMPLES**
pN or p[N]	Both refer to the current (i.e., most recently assigned) value of note parameter N. If the brackets are used, N may be any valid cmusic expression.	p5, p[6], p[p6+3]
vN or v[N]	Both refer to the current (i.e., most recently assigned) value of global variable N. If the brackets are used, N may be any valid cmusic expression.	v1, v[v1], v[2*v[p8-1]-1]
NUMBERS	*Numerical values may be written in any of three number bases (decimal is the default). All numerical values are of type single precision float whether they include a decimal point or not.*	
decimal	Any string of the digits 0 through 9, optionally including a decimal point.	3, 3.1415926535
octal	Any string of the digits 0 through 7 which does not include a radix point and which begins with the digit 0.	077, 03700
hexadecimal	Any string of the digits 0 through 9 plus the letters a through f which does not include a radix point and which begins with the characters 0x.	0x3f, 0x10
OPERATORS	**DESCRIPTION**	**EXAMPLES**
PARENTHESES	*Parentheses must balance, and may be used freely to establish operator precedence. Function arguments should be enclosed in parentheses.*	2*(p5+3), v[p9/(p8-4)]
UNOPS	*Unary operators are normally done before binary operators, which are normally done before postoperators. The following unary operators are given in order of precedence i.e., functions are evaluated before the unary minus operator.*	
sin, cos, atan, ln, exp, floor, abs, sqrt, rand	The standard sine, cosine, and arctangent functions from the UNIX math library, plus natural logarithm, exponential, floor, absolute value, and square root functions. rand returns a random value between 0 and its (positive) argument.	sin(45Deg), ln(p6)-ln(p5), 3+rand(7-3)
-	Unary minus operator.	-3+p7
BINOPS	*Binary operators are evaluated in the precedence order shown below—any operator in the first braced-enclosed set is evaluated before any operator in the next brace-enclosed set, etc., unless parentheses are used to specify otherwise.*	
{^, %}	*a*^*b* means *a* to the *b* power; *a*%*b* means *a* modulo *b*.	3^.5, 2^(1/12), p7%2
{*, /}	*a***b* means *a* multiplied by *b*; *a*/*b* means *a* divided by *b*.	p5*exp(1), ln(p6/v2)
{+, -}	*a*+*b* means *a* plus *b*; *a*-*b* means *a* minus *b*.	p5+exp(1), ln(p6-v2)
POSTOPS	*Postoperators are normally evaluated last (after all other operators have been evaluated). They generally modify the value of the entire expression that precedes them. In the examples that follow, ξ stands for any valid cmusic expression.*	
ξHz	The Hz postoperator treats ξ as a frequency expressed in Hz and converts it to an oscillator increment, i.e., ξHz means ξ**L*/*R*, where *L* is the current default wavetable length and *R* is the current default sampling rate.	440Hz, 440*2^(1/12)Hz
ξsec, ξms	The sec (ms) postoperator treats ξ as a duration expressed in seconds (milliseconds) and converts this value to an oscillator increment, i.e., ξsec means ξ**R*/*L* = 1/ξHz (and ξms means ξ**R*/(1000**L*)).	p4sec, 79ms
ξS	The S postoperator treats ξ as a duration expressed as a number of samples at the current sampling rate and converts this value to a number of seconds, i.e., ξS means ξ/*R*.	16384S
ξdB	The dB postoperator treats ξ as an amplitude value expressed in decibels with a reference value of 1.0 (= 0dB) and converts this value to a linear amplitude, i.e., ξdB means 10^(ξ/20).	0dB, p5/7dB
ξK	The K postoperator multiplies the value of ξ by 1024.	16K, 48/2K
ξDeg	The Deg postoperator treats ξ as a number of degrees and converts this value to the equivalent number of radians, i.e., ξDeg means ξ2π/360.	sin(72Deg)
ξMM	The MM postoperator treats ξ as a Maelzel metronome tempo indication and converts this value to the number of seconds per beat at the specified tempo, i.e., ξMM means 60/ξ.	72MM
ξIS	The IS postoperator computes the sum of the first ξ inverse integers, i.e., 3IS means 1/1+1/2+1/3. 0IS = 0 by definition.	79IS

2.3. The C Preprocessor

In addition to expressions, cmusic provides macro and include-file capabilities by first feeding the score through the C language preprocessor.

Macros allow the user to define a private shorthand, and often save a lot of typing. The simplest form of macros involves straight text substitution. If we include the statement

```
#define A 440Hz
```

in the score, any time we type "A" thereafter will cause cmusic to read "440Hz". Note that no semicolon is used—the *end of the line delimits the end of the definition*. Multiline definitions are possible by using a backslash (\) before the end of a line that comes before the end of a definition, as in

```
#define A \
440Hz
```

Macros can also have *arguments*. If we write the definition

```
#define HORN( start, end ) note start horn end-start
```

then we can play notes on instrument "horn" with statements such as

```
HORN( 1, 2 ) -20dB 440Hz ;
```

which will be read by cmusic as

```
note 1 horn 2-1 -20dB 440Hz ;
```

In this example, the first argument has been given the value 1, and the second argument has been set to 2, causing the macro to expand in the manner shown above. This definition changes the way in which timing is expressed from a starting time and a duration to a starting time and an ending time—the expression within the macro definition takes care of the minor computation needed to do this.

Care must be exercised to ensure that blanks do not creep into expression fields. For example, the definition

```
#define HORN( start, end ) note start horn end - start
```

wouldn't work because of the blanks surrounding the minus sign.

The other type of statement processed by the preprocessor has the form

```
#include "filename"
```

If this were typed in a cmusic score, the entire contents of the named file would be read in and substituted for the "#include" statement. It is generally necessary to give the full pathname of the file unless it is certain that the file will always be in the same directory as the one in which cmusic is run.

A file containing many "standard" cmusic definitions can be accessed with the special statement

```
#include <carl/cmusic.h>
```

The contents of this file can be examined directly to ascertain the nature of the definitions it contains. The one currently in use at CARL is quite lengthy, so only a portion of it is reproduced below.

```
/* cmusic.h - standard macro definitions for cmusic scores */
/*
 * unit generator statement abbreviations
 */
#define NRES(out,in,gain,cf,bw) nres out in gain cf bw d d d d d d d d d d
/*
 * waveform components
 */
#define PLS(num) num,1,0
#define SAW(num) num,1/num,0
#define TRI(num) num,1/num^2,0
/*
 * useful signal waveforms
 */
#define SINE(func) gen p2 gen5 func 1 1 0
```

```
#define COS(func)  gen p2 gen5 func 1 1 90Deg
#define TRIANGLE(func) gen p2 gen5 func TRI(1) TRI(3) TRI(5) TRI(7); NORM(func)
#define SQUARE(func) gen p2 gen5 func SAW(1) SAW(3) SAW(5) SAW(7); NORM(func)
#define SAWTOOTH(func) gen p2 gen5 func SAW(1) SAW(2) SAW(3) SAW(4) SAW(5)\
SAW(6) SAW(7) SAW(8); NORM(func)
#define PULSE(func) gen p2 gen5 func PLS(1) PLS(2) PLS(3) PLS(4) PLS(5)\
PLS(6) PLS(7) PLS(8); NORM(func)
/*
 * use envelope waveforms
 */
#define ENV(func) gen p2 gen4 func 0,0 -1 .1,1 -1 .8,.5 -1 1,0
#define SLOWENV(func) gen p2 gen4 func 0,0 -1 1/3,1 -1 2/3,.5 -1 1,0
#define PLUCKENV(func) gen p2 gen4 func 0,0 -1 .005,1 -2 1,0
/*
 * gen statement abbreviations
 */
#define GEN0(func) gen p2 gen0 func
#define NORM(func) gen p2 gen0 func 1
#define GEN1(func) gen p2 gen1 func
#define GEN2(func) gen p2 gen2 func
#define GEN3(func) gen p2 gen3 func
#define GEN4(func) gen p2 gen4 func
#define GEN5(func) gen p2 gen5 func
#define GEN6(func) gen p2 gen6 func
#define CHUBBY(func) gen p2 chubby func
#define CSPLINE(func) gen p2 cspline func
#define GENRAW(func) gen p2 genraw func
#define QUADGEN(func) gen p2 quad func
#define SHEPENV(func) gen p2 shepenv func
/*
 * general period definition
 */
#define P (p4sec)
/*
 * pitch reference = middle C
 */
#define REF (220*2^(3/12))
/*
 * 12-tone temperament frequencies
 */
#define FR(pitch,oct)(REF*2^(oct)*2^(pitch/12))
/*
 * tempered scale pitch classes (0 octave = middle C up to B)
 */
#define C(oct)(FR(0,oct)Hz)
#define Cs(oct)(FR(1,oct)Hz)
#define Df(oct)(FR(1,oct)Hz)
#define D(oct)(FR(2,oct)Hz)
#define Ds(oct)(FR(3,oct)Hz)
#define Ef(oct)(FR(3,oct)Hz)
#define E(oct)(FR(4,oct)Hz)
#define F(oct)(FR(5,oct)Hz)
#define Fs(oct)(FR(6,oct)Hz)
#define Gf(oct)(FR(6,oct)Hz)
```

```
#define G(oct)(FR(7,oct)Hz)
#define Gs(oct)(FR(8,oct)Hz)
#define Af(oct)(FR(8,oct)Hz)
#define A(oct)(FR(9,oct)Hz)
#define As(oct)(FR(10,oct)Hz)
#define Bf(oct)(FR(10,oct)Hz)
#define B(oct)(FR(11,oct)Hz)

#define RAND(low,high)(rand(high-low)+low)
#define xy(distance, direction) distance*cos(direction) distance*sin(direction)
#define LOG10(x)(ln(x)/ln(10))
#define LOG2(x)(ln(x)/ln(2))
```

Among the useful definitions on this file are shorthands for wavetable generation, certain unit generator definitions, mathematical operations, and pitch class names.

With these definitions in effect we may type, for example, "SINE(f1)" to fill wavetable *f* 1 with a sine waveform, "RAND(9,13)" to choose a random value between 9 and 13, "LOG2(p7)" to take the logarithm of *p* 7 to the base 2, or C(0) to specify the frequency of middle C.

According to the definitions above, Cs(1) expands into

```
(FR(1,1)Hz)
```

Since FR is also a defined macro, this expands further into

```
((REF*2^(1)*2^(1/12))Hz)
```

which again expands into

```
(((220*2^(3/12))*2^(1)*2^(1/12))Hz)
```

This expression calculates the oscillator increment associated with the precise frequency of C-sharp a minor ninth above middle C in equal tempered tuning based on *A* 440. The octave numbers change on octaves of the pitch class C, i.e., B(–1) is a semitone lower than C(0).

We could easily transpose an entire score that uses these macros simply by redefining the reference pitch. Other modifications to the standard definitions are easily implemented.

2.4. Command Descriptions

2.4.1. set

The cmusic set command has two basic forms:

set *parameter* = *value* ;

and

set *option* ;

Some of the options available through the set command are also available as command line flags. In case a score set statement and a command line flag conflict, the command line flag always takes precedence.

set barefile = filename ;

Produces a "bare" score listing on the named file. A "bare" score has all macros and expressions replaced with their numerical values [Default: no barefile is produced].

set blocklength = N;

Sets the i/o block length to N [Default: N = 256] (in command flag form, use –BN).

set channels = N; set nchannels = N; set stereo; set quad;

Determines the number of output channels [Default: N = 1].

set cutoff = N;

Sets cutoff threshold for computing the tail of the global reverb to N [Default: N = .0001 (= –80dB)]. This option affects the space unit generator only.

set floatoutput; set nofloatoutput;

Specifies whether cmusic should produce 32-bit floating point sample values [this is the default]. If floats are not produced, 16-bit 2's complement short ints containing fixed-point binary fractions are produced.

set functionlength = N;

Sets the default score function (wavetable) length to N [Default: N = 1024] (in command flag form, use –LN).

set header; set noheader;

Specifies whether a csound header should be produced by cmusic (such headers are normally transparent to the user). [The default is to produce the header.]

set listingfile = filename;

Produces a score listing on the named file [Default: no listing is produced]. If this command is given with no list filename specified, cmusic will create a file with the same first name as the score file with a ".list" extension.

set noclip;

Turns off clipping of cmusic output (allows float output to exceed range of –1 to +1) [Default: clipping is turned on]

set notify;

Specifies that cmusic should produce its termination message on the terminal (in command flag form, use –n; the –q flag turns off any notify, timer or verbose options in the score).

set outputfile = filename;

Causes cmusic to place its sample output on the named file instead of stdout.

set rand = N; set seed = N;

"Seeds" the random number generator with N.

set sfbuffersize = N;

Sets the buffer size used by the sndfile unit generator to N. This option affects the sndfile and splice unit generators only. [Default: N = 4K (bytes)].

set rate = N; set samplingrate = N; set srate = N;

Set the sampling rate to N [Default = 16K = 16384] (in command flag form, use –RN).

set revscale = N;

Sets relative amplitude of global reverb to N [Default = .25]. This option affects the space unit gen-

erator only.

set room = Lx1,Ly1 Lx2,Ly2 Lx3,Ly3 Lx4,Ly4;

Sets the quadrant 1, 2, 3, and 4 vertices of the inner room to the specified values (given in meters, with point 0,0 being the center of all space) [Default: 4,4 –4,4 –4,–4 4,–4] This option affects the space unit generator only.

set space = Ax1,Ay1 Ax2,Ay2 Ax3,Ay3 Ax4,Ay4;

Sets the quadrant 1, 2, 3, and 4 vertices of the outer room to the specified values (given in meters, with point 0,0 being the center of all space) [Default: 50,50 –50,50 –50,–50 50,–50] This option affects the space unit generator only.

set speakers = Sx1,Sy1 Sx2,Sy2 ... Sxn,Syn;

Sets the locations of n speakers (must be on the perimeter of the inner room) the specified values (given in meters, with point 0,0 being the center of all space) [Default: 4 speakers defined at 4,4 –4,4 –4,–4 4,–4] This option affects the space unit generator only.

set tempowith vN; set offsetwith vN;

Causes cmusic to use the specified variable to set the tempo or time offset. If tempo is specified with, say, v1, then the values of both p2 and p4 will be REPLACED with p2*v1 and p4*v1, respectively. If time offset is specified with, say, v2, then v2 will be added to all p2 values after they have been scaled by tempo, if any. For example:

```
#define TEMPO set tempowith v10 ; var p2 v10
#define T (p2+p4)/(v10)
TEMPO 30MM;
note 0 one 1 440Hz 0dB ;
note T one 1 440Hz 0dB ;
note T one 1 440Hz 0dB ;
```

will use variable v10 to set the tempo of the score. It is set up and defined with macro TEMPO. Note the use of v10 in the T macro.

set t60 = N;

Sets the global reverberation time to N seconds (approximately) This option affects the space unit generator only.

set timer;

Cause cmusic to send computation timing numbers to the user's terminal. When 1 second of score is completed, ":1(0.78563)" is printed, etc. The number is parentheses is the maximum amplitude generated during that second. A number in square brackets following the timing value (such as ":5(1.284562)[127]") indicates the number of out-of-range samples during that second of output (in command flag form, use –t; the –q flag turns off any notify, timer or verbose options in the score).

set verbose;

Causes cmusic to send the interpreted input listing, timing, and termination notification messages to the user's terminal (in command flag form, use –v; the –q flag turns off any notify, timer or verbose options in the score).

2.4.2. ins

The ins command specifies that unit generator statements defining a cmusic instrument are to follow. The command has the general form

ins *time name* ;

where *time* is the action time at which the instrument is to be defined (this is normally 0), and *name* is the name given by the score-writer to the instrument.

All instruments have definitions of the following general form

ins *time name* ;
unit generator statements
end ;

The unit generator statements will be discussed in detail later on.

2.4.3. end

The end command is used only as shown above to terminate an instrument definition.

2.4.4. gen

The gen command is used to invoke one of the available wavetable generators. It has the general form

gen *time name function parameters* ;

where *time* is an action time (this is normally 0), *name* is the name of the wavetable-generating program that the score-writer wishes to invoke, *function* is the name of a cmusic wavetable (e.g., f1, f2, etc.), and *parameters* are as required by the particular program invoked.

The generator programs will be discussed in detail later on.

2.4.5. note

The note command specifies that a note event is to generated. It has the general form

note *time instrument duration parameters* ;

where *time* is the starting time of the note event, *instrument* is the name of the instrument to be played, *duration* is the duration of the note event in seconds, and *parameters* are as required by the particular instrument being played.

All times and durations are specified in seconds. In order to be playable, the list of notes must be given in non-decreasing order of starting times (but see merge, below). If the action time of *any* command is less than one already encountered, cmusic will notify the user of a sequence error and stop.

An important concept in cmusic is that of note parameters, or *p-fields*. Each *field* of a note statement is considered to be a *note parameter field*. Note parameters are numbered according to field positions, so the first field (the "note" command itself) is considered to be p1, the second field (the starting time) is p2, the instrument name is p3, and the duration is p4. The first four p-fields always have these fixed meanings. Subsequent p-fields are given according to the needs of the instrument being played.

Since p2 and p4 are numerical values, they may be cmusic expressions. Further, cmusic expressions may refer directly to p-field values simply by naming them as part of an expression.

A common device is to specify either "p2" or "p2+p4" as an action time. If "p2" is used, then the event will occur at whatever time was last specified as the action time of any previous cmusic statement. If "p2+p4" is used, then the action will occur as soon as the previous note is finished, since p4 always refers to the *last* value to which p4 was set. For example, the following notes will be played in sequence, each with a duration of one second.

```
note     0 toot 1 ... ;
note p2+p4 toot 1 ... ;
note p2+p4 toot 1 ... ;
note p2+p4 toot 1 ... ;
note p2+p4 toot 1 ... ;
```

Technically it would be possible (but bad form) to specify "p2+p4" as the starting time of the first note in this example, since all p-fields initially have the value 0.0.

The following example illustrates a simple way to specify two-note chords in sequence.

```
note     0 toot 1 ... ;
note p2    toot 1 ... ;

note p2+p4 toot 1 ... ;
note p2    toot 1 ... ;

note p2+p4 toot 1 ... ;
note p2    toot 1 ... ;
```

The first two notes have the same starting time (and duration), the next two notes will both start at time 1, the third pair of notes will start at time 2, and so on.

2.4.6. sec

The sec command has the general form

```
sec time ;
```

where *time* is the time at which the section is to be terminated. The *time* field may be *omitted*, in which case cmusic will automatically terminate the section after the last sounding note has ended. If *time* is later than the end of the last sounding note, cmusic will generate the requisite amount of *silence* needed to terminate the section at the specified moment.

Once the section is terminated, the action time of the score is reset to 0.0 and score processing continues. This allows major sections of a cmusic score to be rearranged easily, repeated, etc. For example, we could repeat those two-note chords of the previous example by interposing a sec statement after each repetition, as in the following example.

```
note     0 toot 1 ... ;
note p2    toot 1 ... ;

note p2+p4 toot 1 ... ;
note p2    toot 1 ... ;

note p2+p4 toot 1 ... ;
note p2    toot 1 ... ;

sec ;

note     0 toot 1 ... ;
note p2    toot 1 ... ;

note p2+p4 toot 1 ... ;
note p2    toot 1 ... ;

note p2+p4 toot 1 ... ;
note p2    toot 1 ... ;
```

2.4.7. var

The cmusic var command is used to set global variable values. Global values are maintained in static (i.e., permanent) arrays. Global variable values may be used in cmusic expressions and in certain unit generator parameters. They all have names like v1, v2, v3, ..., for numerical values and s1, s2, s3, ..., for string values.

The general form of the var command is

var *time variable value(s)* ;

where *time* is an action time, *variable* is the name of a global variable (either v-type or s-type), and *value(s)* stands for one or more values to be assigned to the named variable and its successors.

For example, we may write

var 0 v1 440Hz ;

to assign the increment associated with the pitch *A*440 to global variable v1 (at time 0). We also could write

var 0 v1 440Hz 880Hz ;

to set v1 to the increment value for *A*440 and v2 to the increment value for *A*880. If more than one value is given they are assigned to contiguously-numbered variables starting with the one named.

var statements act like assignment statements in programming. Macros may be used to dub variables with more imaginative and revealing names. For example, we might use global variables to obtain random selections from a particular pitch set.

```
#define PITCH(index) v[index]
#define PSET(position) var p2 PITCH[position]
#define S1 1
#define S2 (S1+5)

PSET(S1) A(0) Cs(1) E(1) F(1) Gs(1) ; {pitch sets have 5 elements today}
PSET(S2) Af(0) C(1) Ef(1) E(1) G(1) ;

note p2+p4 honk 3 PITCH(S2+rand(5)) ... ;
```

S1 and S2 define the starting indices in the variable array for two pitch sets, each of which contains five arbitrary pitches. The pitch specified in p5 of the note statement in the example would be selected randomly from the second set of pitches given.

String variables work in the same way, except the values are strings instead of numbers. For example the statement

var 0 s1 "file1" "file2" ;

would set the value of s1 to the first string given, and the value of s2 to the second string given.

2.4.8. merge

Sometimes it is useful to write scores—especially notelists—out of time order. For example, the score-writer may wish to write all of the notes for one musical voice, then all of the notes for a second voice, and so on. Or, it may be desirable to give note statements not in time order for some reason. Perhaps the action times of notes are random (i.e., computed with the rand function), and the proper order of notes literally cannot be determined in advance.

In order to accommodate such scores, the *merge* command may be used. The merge command reads in one or more merge sections, evaluating p2 and p4 fields as it goes. It then sorts each section into order, and merges the sections together into a single, sorted score.

The merge command operates by reading all score statements until an *endsec* statement is encountered. An *endmerge* statement must follow the final endsec statement in each merged portion of the score. During this provisional reading of the score, all var statements are provisionally executed, since p2 (starting time) and

p4 (duration) in note statements might depend on the values of some v-type variable. The p2 and p4 fields of all note statements are evaluated (in case they are expressions—even random ones!), and replaced by numerical constants. The p2 fields on all gen and var statements are also evaluated.

All statements up to the endsec statement are placed into a special temporary file. More than one endsec-delimited section may be included before the endmerge statement. A new temporary file is created for each merge section given before the endmerge statement. Any ins, merge, sec, set or ter statements are treated as *errors* within a merge.

After the temporary files have been created, they are individually sorted into nondecreasing time order, using the constant numbers in the p2 fields as keys for the sort. Then the files are merged together, so that cmusic plays the notes in the correct order.

The following notelist shows two merge examples. In the first merge, notes with random starting times are placed into time order for performance by cmusic. In the second merge, two sections are merged together so that they are performed simultaneously. Note that it would be impossible to play either part of this score if the merge command did not exist, because the action times are unpredictable.

```
merge; {play the following random pitches at random times for 10 seconds}
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz;
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz;
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz;
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz;
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz;
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz;
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz;
    note 0+rand(10) toot rand(1) -23dB 220+rand(880)Hz; endsec; endmerge;

merge; {play the following two merge sections simultaneously}
    note 1 toot rand(1)+.2 -23dB A(rand(5)-3)+rand(12);
    note p2+p4 toot rand(1)+.2 -23dB A(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB A(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB A(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB A(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB A(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB A(1+rand(12)); endsec;
    note 3 toot rand(1)+.2 -23dB C(rand(5)-3)+rand(12);
    note p2+p4 toot rand(1)+.2 -23dB C(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB C(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB C(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB C(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB C(1+rand(12));
    note p2+p4 toot rand(1)+.2 -23dB C(1+rand(12)); endsec; endmerge;
```

2.4.9. ter

The ter command has the general form

ter *time* ;

where *time* is the time at which the score is to be terminated. The *time* field may be *omitted*, in which case cmusic will automatically terminate the score after the last sounding note has ended. If *time* is later than the end of the last sounding note, cmusic will generate the requisite amount of *silence* before terminating the score.

A combination of sec and ter statements can be used to supply a few seconds of silence at the end of a score:

```
note ...
note ...

sec ; {finish playing all notes and reset action time to 0}

ter 3 ; {pad with 3 seconds of silence for reverb to die away}
```

This is often necessary when the *space* unit generator is in use in order to allow the accumulated reverberation to die away.

2.5. Wavetable Generation

A growing library of free-standing programs exists for generating wavetable values. cmusic users may write their own generating programs if they do not find a suitable one already in the following list.

cmusic Wavetable-generating Programs

NAME	*DESCRIPTION*
chubby	Chebychev polynomial function generator
cspline	smooth curve interpolator
gen0	normalize a function to lie within the range +max to -max:
gen1	straight line segment function generator
gen2	Fourier synthesis function generator
gen3	simple line segment function generator
gen4	exponential curve segment generator
gen5	Fourier synthesis generator
gen6	random table generator
genraw	read wavetable values from a file
quad	sound path interpreter
shepenv	generator for Shepard tone spectral envelopes

cmusic invokes ones of these generating programs with the gen command. For example, the command

```
gen 0 gen3 f2 0 1 1 0 ;
```

instructs cmusic to execute the following UNIX-level command:

```
gen3 -L1024 0 1 1 0
```

The output of the command is a set of 1024 wavetable values—cmusic places these into wavetable *f* 2 (cmusic uses the *popen* facility of UNIX and C to do this). The "-L" flag informs the invoked program how many values it is to generate (this is usually the same as the current default wavetable length).

Certain gen programs make a distinction between functions that are *open* and *closed* on the right. If so, they accept a "-o" flag to specify open and "-c" to specify closed. If no such flag is given, the program is free to use a default.

In practical terms, open functions are used periodically—such as a waveshape—and closed functions are used once only—such as an amplitude envelope—over the course of a single note.

A function is open on the right if its last value is not exactly the same as its first. The wavetable shown in section 1.8.3. illustrates an open function. If the last value of this wavetable were the same as the first, then the shape of the periodic waveform would be incorrect when it is used repetitively by a cmusic osc unit generator.

On the other hand, an amplitude envelope typically will start and end with a zero value since we wish the envelope not to produce a click at either end of the note. Such a function is said to be closed on the right.

Mathematically, the difference between an open and a closed interval may be understood as follows. Let t_1 and t_2 be two time values, with $t_1 < t_2$. The interval $t_1 < t < t_2$ is called open at both ends, while the

interval $t_1 \leq t \leq t_2$ is said to be closed on both ends.

Brackets and parentheses are often used to express this difference as well, with (t_1, t_2) referring to the open interval and $[t_1, t_2]$ referring to the closed interval. It is possible for an interval to be closed only on one end and open on the other, notated as $[t_1, t_2)$, which refers to the precise interval $t_1 \leq t < t_2$. In these terms, the gen program flag "–c" specifies a function generated over an interval that is closed at both ends, and the "–o" flag specifies a function generated over an interval that is closed on the left and open on the right.

In particular, for a function that has an intrinsic period of 2π and a wavetable length L, the open version of the function is defined over the interval $i2\pi/L$ for $i = 0, 1, ..., L-1$, while the closed version is defined over the interval $i2\pi/(L-1)$ for $i = 0, 1, ..., L-1$.

gen programs are written so that they may be invoked directly as UNIX commands, in which case they display print out the numbers on the user's terminal. These numbers may be further processed to produce graphs, for example, allowing the score-writer to experiment with various parametric values in the process of writing a score. Like cmusic, though, when the output of a gen program is not connected to a terminal, it is produced as a stream of single precision floating point values intended (usually) for storage in a cmusic wavetable.

A final point is that cmusic allows the score-writer to type expressions and even string variables as parameters to be passed on to the gen program. cmusic *always* includes the –L flag with the number of function values to be generated, but further program flags may be specified in the score as shown in the following example.

```
var 0 s3 "–o –x" ;
GEN3(f2[256]) s3 0 –6dB –12dB 0 ;
```

cmusic translates the expressions into numerical values and includes the value of the string variable in the command as well. The command created by cmusic from the score statements above would look approximately like this:

```
gen3 –L256 –o –x 0 .5 .25 0
```

This example illustrates that the cmusic gen statement will accept a *nonstandard function length* if it is enclosed in brackets after the function name, that cmusic translates expressions into numerical values, and that string values may be included anywhere in the parameter list. Of course, it is up to the invoked program to interpret the command.

Armed with this general background, we now can discuss each of the gen programs in turn.

2.5.1. chubby

The general form of the cmusic statement for this gen program is

gen *time* chubby *function* DC A1 A2 A3... AN

chubby[19] is a gen program useful for generating functions for "waveshaping", or nonlinear distortion techniques. Such functions are normally used in conjunction with the *lookup* unit generator (discussed later).

The arguments specify the amplitude of each of the partials of the Chebychev polynomial function according to the recursion relation

$$T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$$

where $T_0(x)$ sets the relative amplitude of the 0 Hz component, $T_1(x)$ controls the fundamental energy, $T_2(x)$ controls the second harmonic, etc. Here is an example that produces 50% fundamental and 50% third harmonic (the resulting function is shown in Figure 27).

```
CHUBBY(f2) 0 .5 0 .5 ;
```

19. The chubby, cspline, genraw and quad programs were written by Gareth Loy.

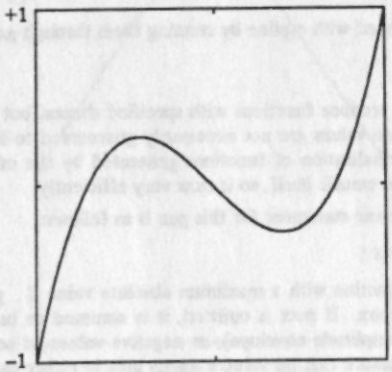


Figure 27: Example of chubby Output (see text).

Whether the function returned by chubby is normalized to the signed unit interval (i.e., ±1) depends on the sum of its arguments being 1.0. It is customary to renormalize chubby functions with gen0.

2.5.2. cspline

The general form of the cmusic score statement is

gen *time* cspline *function* x0 y0 x1 y1 ... xN yN ;

cspline takes pairs of numbers as abscissas and ordinates of a function to be smoothed by a cubic spline function. It produces a smoothed function that includes the input values.

Here is a sample statement in a cmusic score (the resulting function is shown in Figure 28):

CSPLINE(f1) 0 0 .1 .1 .2 1 .3 0 ;

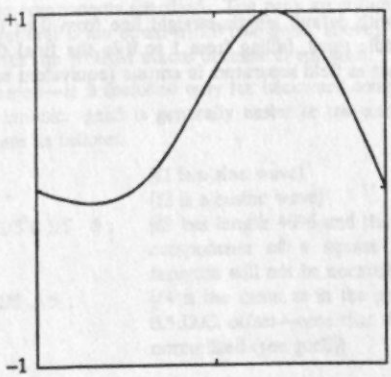


Figure 28: Example of cspline Output (see text).

It is possible that cspline will produce y values that will exceed those supplied; therefore it is a good

idea to normalize functions produced with cspline by running them through gen0.

2.5.3. gen0

Most cmusic gen programs produce functions with specified shapes, but the functions are often in need of *normalization*, i.e., their extreme values are not necessarily guaranteed to lie between plus and minus one. gen0 is provided to expedite normalization of functions generated by the other gen programs. gen0 is the only gen program that is built into cmusic itself, so it runs very efficiently.

The general form of the cmusic statement for this gen is as follows:

gen *time* gen0 *function* max ;

function is a *previously defined* function with a maximum absolute value X. gen0 scales this function so that the old value X corresponds to max. If max is omitted, it is assumed to be 1.0. Some functions consist of only positive values (such as an amplitude envelope), or negative values or both positive and negative values (such as a waveform). gen0 will insure that no value exceeds max in either case. For example,

```
GEN0(f1) 1 ;  { may also be written as NORM(f1) ; }
GEN0(f2) 1000Hz ;
```

The first statement normalizes maximum value of f1 to 1.0, and the second normalizes f2 so that its maximum value corresponds to 1000 Hz.

2.5.4. gen1

The general form of the cmusic statement for this gen program is

gen *time* gen1 *function* t1 v1 t2 v2 ... tN vN ;

gen1 generates a (closed) *function* which starts with value $v1$ at point $t1$ (the beginning of the function), continues in a straight line to value $v2$ at point $t2$, etc., until final value vN is reached at the end of the function.

As for all such gen programs, the specification of t-values goes from 0 to an arbitrary positive value—often taken to be 1. The gen program "maps" these t-values into the number of points required by the –L flag. For example, the statement

```
GEN1(f2) 0,0 1/3,1 2/3,1 1,0 ;
```

generates a trapezoidal function with default length; straight line from 0 to 1 for the first third of the function, level value of 1 for the middle third, falling from 1 to 0 in the final third (see Figure 29). Note the clarifying use of commas, which act as field separators in cmusic (equivalent to blank spaces).

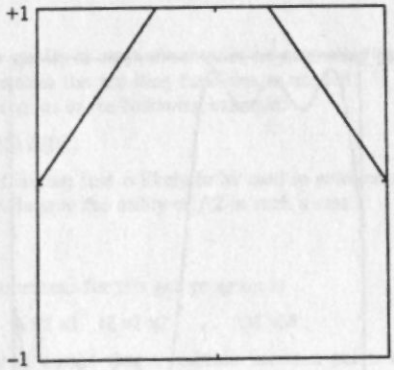


Figure 29: Example of gen1 Output (see text).

2.5.5. gen2

The general form of the cmusic statement for this gen program is as follows:

gen *time* gen2 *function* a1 a2 ... aJ b0 b1 ... bK J ;

aM is the amplitude of harmonic *M* in sine phase, (the fundamental frequency, corresponding to a full period of the function, corresponds to *M* = 1), and *bM* is the amplitude of harmonic *M* in cosine phase. *J* is the number of *aM* terms given (gen2 uses this number to distinguish the *a* coefficients from the *b* coefficients). gen2 operates according to the relation

$$W(i) = \sum_{n=1}^{J} a_n \sin\frac{2\pi ni}{L} + \sum_{n=0}^{K} b_n \cos\frac{2\pi ni}{L}$$

where *i* goes from 0 to *L*−1 (*L* is the wavetable length).

Note that the first cosine component is at 0 Hz ("D.C."). The shape of the final (open) function is determined by the sum of all the components specified. The peak amplitude of the final function depends on the coefficients chosen and in general is *not* equal to 1.0 (see gen0, above). The simplest way to normalize a function generated by gen2 is with the NORM macro defined in cmusic.h.

The use of gen2 is deprecated—it is included only for backward compatibility with the MUSIC V program, which is an ancestor of cmusic. gen5 is generally easier to use and accomplishes the same thing as gen2. Examples of gen2 usage are as follows.

```
GEN2(f1) 1   1 ;                    {f1 is a sine wave}
GEN2(f2) 0 1   0 ;                  {f2 is a cosine wave}
GEN2(f3[4096]) 1 0 1/3 0 1/5   5 ;  {f3 has length 4096 and the shape of the first five
                                     components of a square wave—note that this
                                     function will not be normalized (see gen0)}
GEN2(f4) 1 0 1/3 0 1/5 .5 5 ;       {f4 is the same as in the previous example with a
                                     0.5 D.C. offset—note that this function will not be
                                     normalized (see gen0)}
```

Since the last two gen statements will result in functions which do not lie in the range ±1, normalization with gen0 will probably be necessary. The unnormalized result of the last statement (which defines *f* 4) is shown in Figure 30.

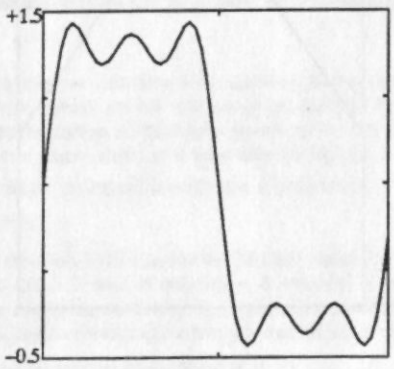


Figure 30: Example of Unnormalized gen2 Output (see text).

2.5.6. gen3

The general form of the cmusic statement for this gen program is

gen *time* gen3 *function* v1 v2 ... vN ;

The arbitrarily long list of values *v*1, ···, *vN* specifies relative amplitude at equally spaced points along a (closed) function. Thus

GEN3 (f2) 0 1 1 0 ;

specifies a trapezoidal function which has 3 parts: the first rises in a straight line from 0 to 1, the second is steady at 1, and the third falls from 1 to 0 (see Figure 31).

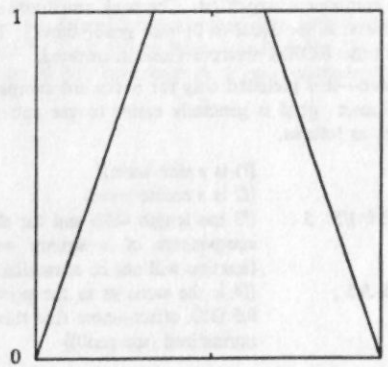


Figure 31: Example of gen3 Output (see text).

If all gen3 parameters are positive, the function is scaled to be only positive. All negative parameters result in an all negative function. Values may be both positive and negative, resulting in a function which

ranges above and below zero.

gen3 functions are easy to specify in ways the require no normalization, but the NORM macro defined in cmusic.h may be used to normalize the resulting functions as needed. In many cases, normalization is not only not necessary, it is not desired, as in the following example.

GEN3(f2) A(0) A(0) B(0) B(0) ;

f 2 is defined here as a control function that is likely to be used to produce a *portamento* on an oscillator unit generator. Normalization would destroy the utility of *f* 2 in such a case.

2.5.7. gen4

The general form of the statement for this gen program is

gen *time* gen4 *function* t1 v1 x1 t2 v2 x2 ... tM vM ;

gen4 operates analogously to gen1 except that transitions between points can be other than straight lines. The t values specify positions along the horizontal axis on an arbitrary scale (as in gen1), the v values specify values of the (closed) function at these points, and the x values specify *transitions* from one point to another. $x = 0$ will yield a straight line, $x < 0$ will yield an exponential transition, and $x > 0$ will yield an inverse exponential transition. If $S_{j,i}$ is the i^{th} function value in the transition from v_j to v_{j+1}, then its shape is determined by the relation:

$$S_{j,i} = v_j + (v_{j+1} - v_j)\frac{1 - e^{i\,x_j/(N-1)}}{1 - e^{x_j}}$$

for $0 \le i < N$, where N is the number of function points between t_j and the next horizontal value. No x value is given after the final point. For example, the statement

GEN4(f2) 0,0 0 1/3,1 0 2/3,1 0 1,0 ;

fills wavetable *f* 2 with a function with straight line transitions between the specified points, as shown in Figure 32.

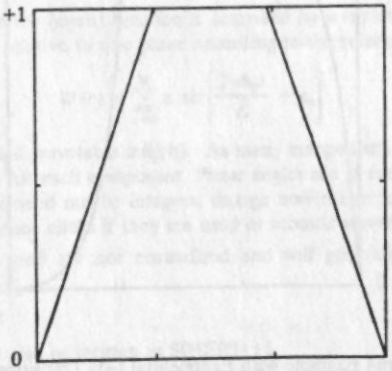


Figure 32: gen4 Example with Straight Line Transitions (see text).

Using a negative transition parameter substitutes exponential transitions between the specified points. For example, the cmusic statement

GEN4(f2) 0,0 –1 1/3,1 0 2/3,1 –1 1,0 ;

would generate the shape shown in Figure 33.

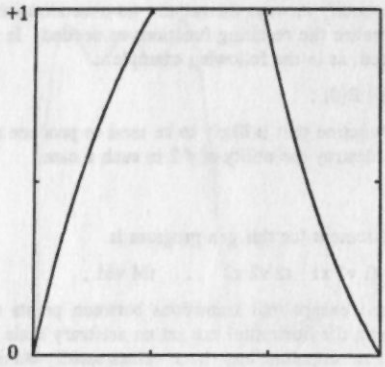


Figure 33: gen4 Example with Exponential (–1) Transitions (see text).

Increasing the magnitude of the (negative) transition parameter increases the curvature of the exponential transitions. For example, the statement

GEN4(f2) 0,0 –5 1/3,1 0 2/3,1 –5 1,0 ;

generates the control function shown in Figure 34.

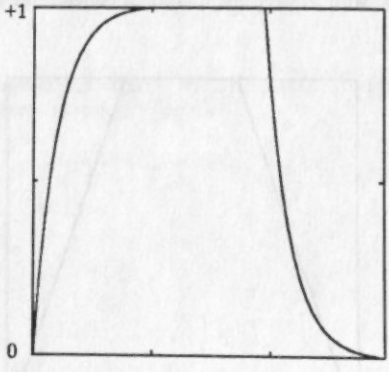


Figure 34: gen4 Example with Exponential (–5) Transitions (see text).

Finally, a *positive* transition parameter specifies *inverse* exponential transitions. For example, the score statement

GEN4(f2) 0,0,10 1/3,1,0 2/3,1,10 1,0 ;

generates a function with the shape shown in Figure 35.

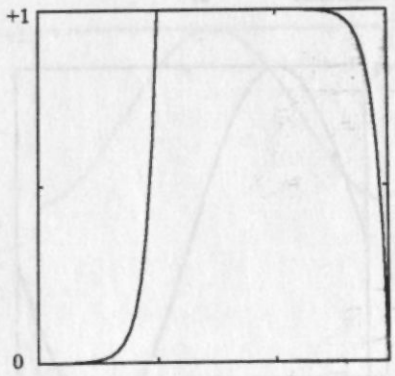


Figure 35: gen4 Example with Inverse Exponential (+10) Transitions (see text).

The transition parameter values specify the number of exponential *time constants* between the endpoints of a transition. As shown in the figures above, a small negative value specifies a curve not very different from a straight line, but which gets near the second value more quickly. A large negative value is more curved, and approaches the second value more quickly. The latter kind of shape is very useful for specifying very sharp attacks while avoiding clicks, for example. Transition types may be mixed freely in a given function specification.

2.5.8. gen5

The general cmusic statement for this gen program has the form

gen *time* gen5 *function* h1,a1,p1 h2,a2,p2 ... hM,aM,pM;

Each Fourier component of the (open) function is described by a triplet specifying a harmonic number, an amplitude, and a phase offset relative to sine phase according to the relation

$$W(i) = \sum_{n=1}^{M} a_n \sin\left(\frac{2\pi h_n i}{L} + p_n\right)$$

for $i = 0$ to $L-1$ (L is the specified wavetable length). As many components may be supplied as desired, but all three values must be supplied for each component. Phase angles are in radians (cf. "Deg" post operator in expressions). Harmonic numbers need not be integers, though non-integer harmonic numbers will generally result in wavetables that will produce clicks if they are used as acoustic waveforms.

Functions generated with gen5 are not normalized and will generally need to be treated with the NORM macro defined in cmusic.h.

For example, the statement

```
GEN5(f1) 1,1,0 ;  { may also be written as SINE(f1) ; }
```

produces a sine waveshape as shown in Figure 36.

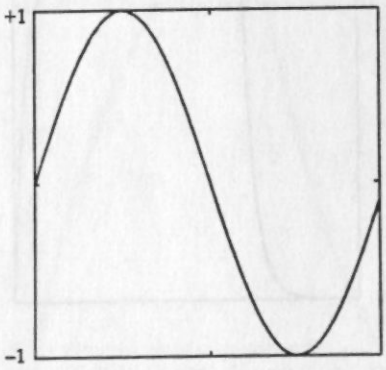


Figure 36: Sine Waveshape Produced with gen5 (see text).

This function already had a specified amplitude and a single component, so no normalization is needed.

However the statement

GEN5(f2) 1,1,0 3,1/3,0 5,1/5,0;

creates a waveshape consisting of the first three components of a square wave. Since more than one component is specified, normalization would be needed to bring the peaks within the range of ±1.0. This function is shown in Figure 37.

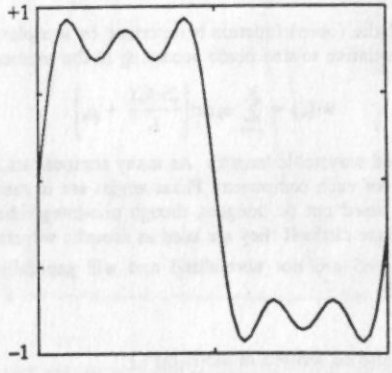


Figure 37: First Three Components of a Square Waveshape Produced with gen5 (Unnormalized).

The phase and amplitude controls of gen5 may be used to create unusual waveshapes as well. For example the statement

GEN5(f3) 1,-.5,90Deg 0,.5,90Deg; {f3 is a raised cosine wave}

generates the "raised," inverted cosinusoidal shape shown in Figure 38.

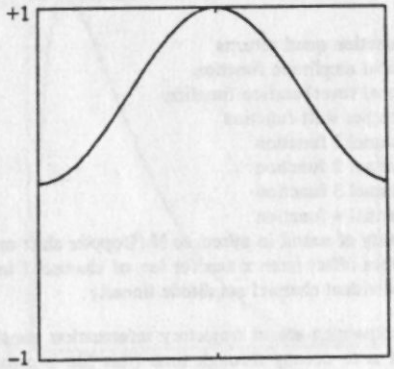


Figure 38: Raised Inverted Cosinusoidal Waveshape Produced with gen5.

2.5.9. gen6

gen6 fills the specified function with uniformly distributed random values in the range -1 to +1. It has a general statement of the form

gen *time* gen6 *function* ;

2.5.10. genraw

genraw reads the named file, which must contain binary floating point numbers (called floatsams at CARL). The floatsams are then written on its standard output. If called from inside cmusic, this will cause the contents of the file to be copied into a function in cmusic. In this way one can use, for example, results of the analysis of natural sounds as waveforms or envelopes in cmusic.

The genraw program is invoked in cmusic with statements such as

```
var p2 s1 "filename" ;
GENRAW(f1) s1 ;
```

genraw will force the number of floatsams it writes to its standard output to equal the number specified by the -L flag by linearly interpolating the number of floatsams in the input file to stretch or shrink the file to fit. This guarantees that cmusic will get as many points as it wants, but may have side effects of altering the character of the function if the stretching or shrinking are extreme.

2.5.11. quad

quad is a special gen program for use in conjunction with the *sndpath* program. sndpath allows the user to design "paths" through space along which a sound source is to seem to move. The paths designed with the sndpath program are read into cmusic with the quad gen program, and the generated wavetables are used to control the *space* unit generator.

The quad program has a general command line form as follows:

quad flags path_file [seconds_duration]

Possible flags are as follows:

–LN	length of function quad returns
–a	produce global amplitude function
–r	produce global reverberation function
–d[N]	produce Doppler shift function
–1	produce channel 1 function
–2	produce channel 2 function
–3	produce channel 3 function
–4	produce channel 4 function
–vN	set the velocity of sound in m/sec. to N (Doppler shift only)
–oN	set the degrees offset from x axis for loc. of channel 1 to N
–t	calculate individual channel amplitude linearly

quad reads the *path_file* containing sound trajectory information consisting of [x,y] pairs representing the succession of points a sound is to occupy through time (this file is created with the sndpath program). quad interprets this path and produces a set of functions that implement the location modulation scheme desired[20].

The first argument to quad specifies the number of points in the cmusic wavetable function (typically 1024). The next argument specifies which of the seven possible functions quad will produce on this invocation of the program. This is followed by the file containing the path, as produced by sndpath.

The syntax for specifying Doppler shift is slightly different since one must also specify the duration of the sound to calculate the correct frequency shift function. In this case, one can either supply the duration as a number concatenated with the –d flag, as in –d5, or provide the value as a separate argument after the pathname. The –v flag provides a way to adjust the velocity of sound, which defaults to 340 m/s.

The –o flag takes a number of degrees offset to use in calculating the location of the speakers. Ordinarily, channel 1 is defined as 45 degrees above the x axis in quadrant 1. The locations of the other channels are computed from this. For instance, –o90 calculates the function for channel 1 as though it were positioned directly in front of the audience, with the other speakers forming a diamond figure around the audience.

cmusic score statements that use quad might be:

```
var p2 s1 "–a " ;
var p2 s2 "pathfile" ;
QUAD(f1) s1 s2 ;
```

Note the blank after the –a in string variable s1, without which the flag would be run together with the pathfile when passed to quad.

2.5.12. shepenv

The general form of cmusic statement used with this gen program is

gen *time* shepenv *function* cycles base ;

shepenv generates a function suitable for use with the *illus* unit generator in the production of Shepard or Risset illusions. The function consists of a "raised inverted cosine" waveform that begins and ends with the value 0.0, and attains a single peak of 1.0 in between (i.e., on a linear scale, $f(x) = -.5\cos(x) + .5$, for $0 \leq x \leq 2\pi$). shepenv scales this function along a logarithmic abscissa, so that equal pitch intervals occupy equal intervals along the horizontal axis. shepenv accepts 2 arguments: the number of logarithmic cycles, and the logarithm base.

For example, if the spectral envelope is to be scaled over 6 octaves, then cycles would be equal to 6, and base would be equal to 2 (see Figure 39).

---

20. The method used is described in detail in John Chowning's "The Simulation of Moving Sound Sources", *Journal of the Audio Engineering Society*, Volume 19, Number 1, January, 1971.

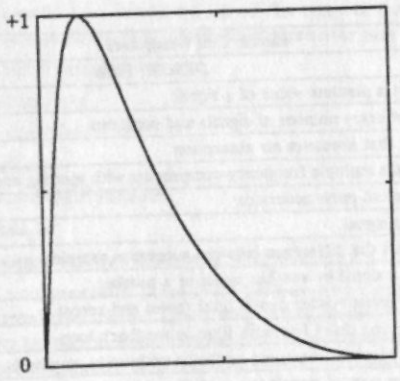


Figure 39: shepenv Output for *cycles* = 6 and *base* = 2 (see text).

If the envelope is to span 7 perfect fifths, then set cycles to 7 and base to 3/2, as shown in Figure 40.

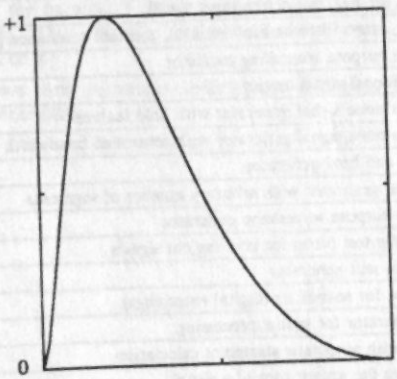


Figure 40: shepenv Output for *cycles* = 7 and *base* = 1.5 (see text).

2.6. Unit Generators

Unit generators are subprograms that perform some signal generation or processing task. The heart of cmusic operation lies in the operation of the unit generators. The following table lists most of the unit generators currently implemented as part of the cmusic program.

cmusic Unit Generators	
NAME	*DESCRIPTION*
abs	calculates absolute value of a signal
adn	adds arbitrary number of signals and constants
airabsorb	a filter that simulates air absorption
band	generates multiple frequency components with spacing and amplitude control
blp	bandlimited pulse generator
delay	delays a signal
diff	calculates the difference between successive samples
div	divides a signal by another signal or a number
flt	general second order digital filter (poles and zeros)
fltdelay	recirculating delay line with filter in feedback loop
freq	general purpose truncating oscillator with unity amplitude
illus	generates control signals for pitch illusions
integer	calculates integer part of a signal
inv	outputs one minus the input signal
iosc	general purpose interpolating oscillator
lookup	general nonlinear transfer function (waveshaper)
mult	multiplies arbitrary number of signals and constants
neg	output the negative of the input signal
nres	2-pole, 2-zero filter with normalized, constant resonance gain
osc	general purpose truncating oscillator
out	multichannel signal output
rah	random noise signal generator with hold feature
ran	random noise signal generator with controlled bandwidth
sah	sample and hold generator
seg	envelope generator with arbitrary number of segments
shape	general purpose waveshape generator
show	debugging test probe for printing out signals
signum	a signum unit generator
sndfile	oscillator for soundfiles (digital recordings)
space	unit generator for spatial processing
splice	sndfile with automatic startpoint calculation
sqroot	calculates the square root of a signal
square	calculates the square of a signal
trans	arbitrary transition generator
version	creates new versions of soundfiles (digital recordings)
zdelay	interpolating dynamic signal delay

Some of these unit generators are extremely simple (like *adn*) while others are quite complex (like *space*). Most of the commonly used unit generators have been highly optimized for calculation speed. In all cases they are written in the C programming language, which means that it is possible to understand the precise operation of a unit generator by examining its source code, and that it is relatively straightforward to add new unit generators to private copies of cmusic (guidelines about how to do this are included in the cmusic source code).

Unit generators are described in terms of their inputs, output(s), and state variable(s). For example, the *osc* unit generator is tersely described with the line

osc output[b] amplitude[bvpn] increment[bvpn] table[fvpn] sum[dpv] ;

This description tells us how the *osc* unit generator may be used in a cmusic instrument definition. The name of the unit generator appears first, followed by its parameters. Each parameter is given a more or less

descriptive name which is followed by a bracketed list of the types of data sources or sinks to which it may be connected. The abbreviations indicate that a particular parameter may be connected to

- *b* signal outputs of other unit generators,
- *p* note statement parameters,
- *v* global numerical variables,
- *s* global string variables,
- *d* dynamically-allocated state variables,
- *f* wavetable functions, or
- *n* numbers (i.e., constant values—these may be expressions).

As a general rule, any unit generator parameter that *may* be a dynamic (i.e., *d*-type) variable *should* be a dynamic variable under normal conditions. This allows cmusic instruments to be re-entrant, which is to say that any number of notes may be played simultaneously on them. Exceptions to this rule occur when there is reason to save the value of state variables between note events, as when a sequence of notes is played on an instrument containing an oscillator and the score-writer wishes the phase (i.e., *sum*) value to be "remembered" at the end of each note so that the next one will pick up where the previous one left off.

Certain unit generators have a variable number of parameter fields. For example, the *adn* unit generator is described as follows:

```
adn out[b] in[bnpv]* ;
```

The asterisk (*) indicates that the parameter that precedes it may be repeated any number of times (it must appear at least once, however). Thus adn may be used to add together as many inputs as desired, any of which may be of type *b*, *n*, *p*, or *d*.

The following table shows terse parameter descriptions—including possible connection types—for the current flock of cmusic unit generators.

Unit Generator Parameter Descriptions	
UG	*PARAMETER DESCRIPTION*
abs	out[b] in[bnpv] ;
adn	out[b] in[bnpv]* ;
airabsorb	out[b] in[b] x[bnpv] y[bnpv] lx[d] ly[d] r[d] scale[d] c1[d] c2[d] xm1[d] xm2[d] ym1[d] ym2[d] ;
band	out[b] amp[bnpv] afac[bnpv] inca[bnpv] incb[bnpv] ifac[bnpv] table[fnpv] sum[dpv] ;
blp	out[b] amp[bnpv] inc1[bnpv] inc2[bnpv] n[bnpv] sum1[dpv] sum1[dpv] ;
delay	out[b] in[bnpv] gain[bnpv] table[fnpv] len[npv] pos[npv] ;
diff	out[b] in[bnpv] temp[dpv] ;
div	out[b] numerator[bnpv] denominator[bnpv] temp[dpv] ;
flt	out[b] in[bnpv] gain[bnpv] a0[bnpv] a1[bnpv] a2[bnpv] b1[bnpv] b2[bnpv] t1[dpv] t2[dpv] t3[dpv] t4[dpv] ;
fltdelay	out[b] dbuf[dpnv] dlen[dpnv] now[dpnv] del[dpnv] coef[dpnv] durcnt[dpnv] noicnt[dpnv] begcnt[dpnv] shrink[dpnv] strtch[dpnv] sdelay[dpnv] oldpit[dpnv] oldval[dpnv] oldin[dpnv] oldout[dpnv] in[bnpv] pitch[bnpv] decay[bnpv] table[fnpv] level[npv] final[npv] onset[npv] place[npv] filtr[npv] noise[npv] stiff[npv] ;
freq	out[b] incr[bnpv] table[fnpv] sum[bdnpv] ;
illus	ampout[b] incrout[b] ampin[bnpv] incrin[bnpv] which[bnpv] ratio[bnpv] table[fnpv] incrmin[bnpv] incrmax[bnpv] ;
integer	out[b] in[bnpv] ;
inv	out[b] in[bnpv] ;
iosc	out[b] amp[bnpv] incr[bnpv] table[fnpv] sum[bdnpv] ;
lookup	out[b] table[fnpv] in[bnpv] min[npv] max[npv] ;
mult	out[b] in[bnpv]* ;
neg	out[b] in[bnpv] ;
nres	out[b] in[bnpv] gain[bnpv] cf[bnpv] bw[bnpv] t1[dpv] t2[dpv] t3[dpv] t4[dpv] t5[dpv] t6[dpv] t7[dpv] t8[dpv] t9[dpv] t10[dpv] ;
osc	out[b] amp[bnpv] incr[bnpv] table[fnpv] sum[bdnpv] ;
out	in[bdnpv]* ;
rah	out[b] amp[bnpv] incr[bnpv] pos[dpv] from[dpv] to[dpv] ;
ran	out[b] amp[bnpv] incr[bnpv] pos[dpv] from[dpv] to[dpv] ;
sah	out[b] in[b] period[bnpv] temp1[dnpv] temp2[dnpv] ;
seg	out[b] amp[bnpv] table[fnpv] sum[dpv] incr[npv]* ;
shape	out[b] table[fnpv] sum[bdnpv] ;
show	in[bdfnpv]* ;
signum	out[b] in[bnpv] ;
sndfile	out[b] amp[bnpv] incr[bnpv] filename[s] channel[npv] startframe[bnpv] endframe[bnpv] pos[bdnpv] ptr[d] ;
space	in[bnpv] nskip[npdv] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] x[bnpv] y[bnpv] theta[bnpv] len[bnpv]* ;
splice	out[b] amp[bnpv] incr[bnpv] filename[s] channel[npv] startframe[bnpv] endframe[bnpv] pos[bdnpv] ptr[d] old[dnpv] older[dnpv] oldest[dnpv] ;
sqroot	out[b] in[bnpv] ;
square	out[b] in[bnpv] ;
trans	out[b]xo[dpv]len[dpv]i[dpv]t[npv]v[npv]a[npv]*t[npv]v[npv] ;
version	out[b] amp[bnpv] incr[bnpv] filename[s] channel[npv] a[npv] b[npv] c[npv] d[npv] tau[npv] Rorig[npv] Forig[npv] ipos[dnpv] fpos[dnpv] lasto[dnpv] ptr[dnpv] sum[dnpv] ;
zdelay	out[b] in[bnpv] maxtime[npv] table[dnpv] tlen[dnpv] pos[dnpv] gain[bnpv] delay[bnpv]* ;

We now discuss the operation of each current cmusic unit generator in turn.

2.6.1. abs

General cmusic statement:

abs out[b] in[bnpv] ;

The output of *abs* is the absolute value of its input.

2.6.2. adn

General cmusic statement:

```
adn out[b] in[bnpv]* ;
```

The output of *adn* is the (algebraic) sum of all its inputs. Any number of inputs may be given.

2.6.3. airabsorb

General cmusic statement:

```
airabsorb out[b] in[b] x[bnpv] y[bnpv]
  lx[d] ly[d] r[d] scale[d] c1[d] c2[d] xm1[d] xm2[d] ym1[d] ym2[d] ;
```

cmusic.h macro definition:

```
#define AIRABSORB(OUT,IN,X,Y) airabsorb OUT IN X Y d d d d d d d d d d
```

airabsorb is a relatively efficient and rarely useful filter that simulates the sound absorption characteristics of air. The filter is essentially lowpass, with parameters adjusted to simulate the nitrogen and oxygen absorption characteristics of air according to the approximation

```
absorption(freq) (in dB/meter) approx freq/100000
```

Thus a 1kHz signal at a distance of 100 meters suffers a 1dB attenuation, while a 10kHz signal at the same distance suffers a 10dB attenuation. Such absorptions are very small for sounds closer than about 50 meters, but they become significant for larger distances. It calculates the distance from coordinate (0,0) to (x,y) and calculates a lowpass filter that it applies to the input signal (all coordinate values are in meters).

2.6.4. band

General cmusic statement:

```
band out[b] amp[bnpv] afac[bnpv] inca[bnpv] incb[bnpv] ifac[bnpv]
  table[fnpv] sum[dpv] ;
```

band outputs several waveforms at once at frequencies in a band corresponding to *inca* through *incb*. If *ifac* is positive, it is *added* repeatedly to *inca* to get each new frequency; if it is negative, its absolute value is *multiplied* repeatedly to get new frequencies (–2 gives all octaves, for example).

The first frequency component has an amplitude specified by *amp*; *afac* is *multiplied* by *amp* for the second component, *multiplied* again for the amplitude of the third, etc. The output has peak amplitude equal to the sum of the amplitudes of the components generated, which depends on the size of the band, their spacing, and *afac*. If *afac* = 1.0, then the output signal has amplitude N times *amp*, where N if the number of components generated.

band has some similarities to *blp* (see below), and the latter is much more efficient.

2.6.5. blp

General cmusic statement:

```
blp out[b] amp[bnpv] inc1[bnpv] inc2[bnpv] n[bnpv] sum1[dpv] sum1[dpv] ;
```

blp generates a bandlimited pulse wave which contains n equal-amplitude sinusoidal components starting at frequency f_1 (specified by *inc1*) and spaced upwards by frequency f_2 (specified by *inc2*) according to the closed form formula:

$$y(n) = \sum_{k=0}^{N-1} \sin(\alpha + k\beta) = \sin\left[\alpha + \frac{(N-1)\beta}{2}\right] \sin\left(\frac{N\beta}{2}\right) \csc\left(\frac{\beta}{2}\right)$$

where $y(n)$ is the output of *blp*, $\alpha = 2\pi n f_1/R$ and $\beta = 2\pi n f_2/R$.

If f_2 is equal to f_1, then a harmonic spectrum will result, i.e., the first n harmonics of f_1 will be

present, all at equal amplitude. *blp* is useful to efficiently generate harmonic-rich waveforms that do not produce foldover as long as the value of n is less than $1 + (R/2 - f_1)/f_2$ (where R is the sampling rate).

blp is discussed in the introduction to this manual (see Figure 15).

2.6.6. delay

General cmusic statement:

```
delay out[b] in[bnpv] gain[bnpv] table[fnpv] len[npv] pos[npv] ;
```

delay output is equal to its input delayed by *length* and scaled by *gain*. *table* should be a function (or a function index) whose length is at least equal to *length*.

An all-zero function of length 39 milliseconds could be created with the command

```
GEN3(f2[39ms]) 0 0 ;
```

Care should be exercised with this unit generator, since the function may not contain two things at the same time. Therefore, instruments containing delays are not reentrant (they can play only one note at a time). Also, it will usually be necessary to re-clear the delay table before each note begins. A macro to do this might read:

```
#define Dnote(Time,Func)GEN3(func) 0 0 ; note Time
```

2.6.7. diff

General cmusic statement:

```
diff out[b] in[bnpv] temp[dpv] ;
```

The *diff* unit generator outputs the differences between successive samples of its input. The *temp* variable is normally a *d*. If the input signal is x(0), x(1), x(2), ..., then output is x(0)–0, x(1)–x(0), x(2)–x(1), ...

2.6.8. div

General cmusic statement:

```
div out[b] numerator[bnpv] denominator[bnpv] temp[dpv] ;
```

The output of the *div* generator is equal to *numerator* divided by *denominator*. Each value of *denominator* is checked to see of it is equal to 0.0, and if so, a huge value is output (it is up to the user to see that this does not happen).

2.6.9. flt

General cmusic statement:

```
flt out[b] in[bnpv] gain[bnpv] a0[bnpv] a1[bnpv] a2[bnpv] b1[bnpv] b2[bnpv]
  t1[dpv] t2[dpv] t3[dpv] t4[dpv] ;
```

flt is a general second-order digital filter that operates according to the input-output relation:

$$y[n] = gain * (a0*x[n] + a1*x[n-1] + a2*x[n-2] + b1*y[n-1] + b2*y[n-2])$$

where *y[n]* is the current output, *y[n–1]* is the previous output, *x[n–1]* is the previous input, etc. The filter requires 5 coefficients, as shown, and 4 temporary locations (*t1* through *t4*), which are normally of type *d*.

The operation of *flt* is discussed in the introduction to this manual.

2.6.10. fltdelay

General cmusic statement:

```
fltdelay out[b]
    dbuf[dpnv] dlen[dpnv] now[dpnv] del[dpnv] coef[dpnv] durcnt[dpnv]
    noicnt[dpnv] begcnt[dpnv] shrink[dpnv] strtch[dpnv] sdelay[dpnv]
    oldpit[dpnv] oldval[dpnv] oldin[dpnv] oldout[dpnv]
  in[bnpv] pitch[bnpv] decay[bnpv] table[fnpv] level[npv] final[npv]
  onset[npv] place[npv] filtr[npv] noise[npv] stiff[npv] ;
```

cmusic.h macro definition:

```
#define FLTDELAY(b) fltdelay b d d d d d d d d d d d d d d
```

where:

in[bvpn]	optional input block (for use as resonator)
pitch[bvpn]	desired fundamental freq (use Hz postop)
decay[bvpn]	duration factor (0 to 1: 1 = full duration)
table[fvpn]	function to initialize table (e.g. from gen6)
level[vpn]	amplitude of pluck (0 to 1: 1 = loudest)
final[vpn]	number of db down at p4 (0 to 100: 40 = norm)
onset[vpn]	attack time for pluck (0 to .1 sec: 0 = fast)
place[vpn]	pluck point on string (0 to .5: 0 = normal)
filtr[vpn]	lowpass prefilter cutoff freq (0 to .5 Srate)
noise[vpn]	time of initial noise burst (–1 to +0.1 sec)
stiff[vpn]	stiffness (0 to 10: 10 = stiffest/sharpest)

fltdelay implements the Karplus-Strong plucked-string algorithm as improved by David Jaffe and Julius Smith[21]. In its simplest form, operation is as follows: *gen6* is used to fill a table with random numbers. The first N random numbers (where N is approximately $R/pitch$) are fed into a delay line of length N. The output of the delay line is the output of the unit generator; but the output is also sent to a filter which computes a weighted average of the current output sample and the previous output sample. The output of this averaging filter is then stuffed back into the beginning of the delay line. The result is that every N samples, a given sample in the loop will be replaced by an average of that sample and the adjacent sample. Hence, the delay line begins with white noise in it and ends up with all zeros (or some small D.C. value) in it. As it happens, the resulting sound can be very similar to that of a plucked string.

Many options can be explored: an input can be connected so that the recirculating delay line acts as a resonator (but beware of overflow—the input should either be low amplitude or the value of *final* should be very large to produce rapid decay). *pitch* can be allowed to vary (but don't let it drop below R/L, where R is the sampling rate and L is the wavetable length). *decay* can be made to vary (but note that the effect of the *decay* parameter depends upon the pitch—higher pitches are attenuated more rapidly for the same value of *decay*). The *final* parameter provides control of the decay independently of pitch (the decay is always exponential, and the duration is always $p4$; *final* specifies whether the note is to be 40 db down by $p4$ or perhaps 80 db down—in which case it will sound shorter). The *onset* parameter applies a linear ramp of duration *onset* seconds to the output of *fltdelay*; this can used to soften the typical pluck attack. It is also possible to output a noise burst of duration *noise* seconds prior to starting up the recirculating delay line. Conversely, a negative value for *noise* causes the recirculating delay line to run for negative *noise* seconds prior to the start of the note. Lastly, a stiffer string can be simulated by specifying sharpened partials (but values of *stiff* less than 4 will probably be inaudible).

The remaining options apply only to the numbers in the delay line just prior to the start of the note: a function other than *gen6* can be used to fill the *table*. The initial amplitude can be set via *level*. A comb prefilter can be applied to the numbers in the *table* to simulate plucking at a different *place* along the string

21. The *fltdelay* and *splice* unit generators were implemented in cmusic by Mark Dolson (also, see *Computer Music Journal* Volume 7, Number 2, pp. 56–69).

(for example, setting *place* to .5 is like plucking the string in the middle in that all even harmonics are eliminated). A lowpass prefilter can be applied to the numbers in the *table* to decrease the brightness of the plucked sound (for example, setting *filtr* to .1 applies a lowpass prefilter with a cutoff frequency of $0.1 \cdot R$). Note: When prefiltering is NOT desired, the value of *filtr* should be .5.

fltdelay can also be used to implement more drumlike sounds by setting *decay* to –1 or letting it switch randomly between +1 and –1 (but don't set it to a value other than plus or minus one unless you want to affect the rate of decay as well).

2.6.11. freq

General cmusic statement:

```
freq out[b] incr[bnpv] table[fnpv] sum[bdnpv] ;
```

A truncating, table lookup oscillator. Dynamic variables are recommended for *sum* locations. *table* may be either explicit, or a number may be given. If the number is n, then $floor(n)$ is used as a wavetable index. NOTE: This unit generator is equivalent to (but faster than) an *osc* with amplitude set to 1.0.

2.6.12. illus

General cmusic statement:

```
illus ampout[b] incrout[b] ampin[bnpv] incrin[bnpv] which[bnpv]
  ratio[bnpv] table[fnpv] incrmin[bnpv] incrmax[bnpv] ;
```

illus is a special Shepard/Risset tone illusion control function generator. It generates amplitude (*ampout*) and frequency (*incrout*) control signals from an input frequency (*incrin*) and a spectral shaping function (*table*) which determines which of M *ratio*-related components in the frequency range *incrmin* to *incrmax* is to be output.

ampout and *incrout* are fed (presumably) to an oscillator that generates a single component of the illusion. Each component has a frequency equal to *ratio* times the last one (unless it exceeds *incrmax*, in which case it wraps around to a frequency between *incrmin* and *incrmax*). The amplitude of each component is obtained from *table* by mapping the table abscissa onto the frequency range [*incrmin*, *incrmax*]. All output amplitudes are also scaled by the constant factor *ampin*.

The following cmusic score generates a full ascending chromatic scale of Shepard tones, using a special gen function for the spectral envelope (*shepenv*) that produces a "raised inverted cosine" envelope over a logarithmic frequency abscissa:

```
#include <carl/cmusic.h>

ins 0 shep ;
   seg        b3 p5 f2 d 0 ;                  {note envelope}
   illus      b1 b2 b3 p10 p6 p7 f3 p8 p9 ;   {spectral envelope}
   osc        b1 b1 b2 f1 d ;                 {carrier}
   out        b1 ;
end ;

{ reserve spectral space for 6 components }
#define NCOMP 6
{ each component separated by an octave }
#define BASE 2
{ spectral enveloped based at 50 Hz }
#define FMIN 50Hz
{ max frequency NCOMP octaves higher }
#define FMAX FMIN*(BASE^NCOMP)
{ amplitude scaler for each component }
#define AMP 1/NCOMP

SINE(f1) ;                                     {component waveshape}
GEN4(f2) 0,0 -3 .1,1 0 .9,1 -1 1,0 ;           {overall note envelope}
SHEPENV(f3) NCOMP BASE ;                       {special spectral envelope}

{macro for tone complex}
{NOTE : NCOMP-1 components fit under spectral envelope}

#define SHEP(time,pitch,dur)\
note time shep dur AMP 1 BASE FMIN FMAX pitch ;\
note p2 shep dur AMP 2 BASE FMIN FMAX pitch ;\
note p2 shep dur AMP 3 BASE FMIN FMAX pitch ;\
note p2 shep dur AMP 4 BASE FMIN FMAX pitch ;\
note p2 shep dur AMP 5 BASE FMIN FMAX pitch ;\
note p2 shep dur AMP 6 BASE FMIN FMAX pitch

SHEP(0,A(-3),.5) ;
SHEP(p2+1,As(-3),.5) ;
SHEP(p2+1,B(-3),.5) ;
SHEP(p2+1,C(-2),.5) ;
SHEP(p2+1,Cs(-2),.5) ;
SHEP(p2+1,D(-2),.5) ;
SHEP(p2+1,Ds(-2),.5) ;
SHEP(p2+1,E(-2),.5) ;
SHEP(p2+1,F(-2),.5) ;
SHEP(p2+1,Fs(-2),.5) ;
SHEP(p2+1,G(-2),.5) ;
SHEP(p2+1,Gs(-2),.5) ;
SHEP(p2+1,A(-2),.5) ; {last note sounds same as first}

ter ;
```

2.6.13. integer

General cmusic statement:

integer out[b] in[bnpv] ;

The output is equal to the integer part of the input.

2.6.14. inv

General cmusic statement:

inv out[b] in[bnpv] ;

The output is equal to 1.0 minus the input.

2.6.15. iosc

General cmusic statement:

iosc out[b] amp[bnpv] incr[bnpv] table[fnpv] sum[bdnpv] ;

An interpolating, table lookup oscillator. Dynamic variables are recommended for *sum* locations. *table* may be either explicit, or a number may be given. If the number is *n*, then *floor(n)* is used as a wavetable index.

iosc requires a much shorter table than *osc* for equivalent distortion but runs more slowly, of course (see the introduction to this manual).

2.6.16. lookup

General cmusic statement:

lookup out[b] table[fnpv] in[bnpv] min[npv] max[npv] ;

lookup is a table lookup generator which uses its input to address the *table* to find the output. The input is clipped to *min* and *max*.

Conceptually, *table* is a function of x, with x in the domain *min* to *max*. The input to *lookup* is then x and the output is $f(x)$. Since *table* is used to provide the mapping from input to output, quite arbitrary correspondences—in particular nonlinear ones—may be specified.

In particular, if $f(x)$ is a sum of Chebychev polynomials an arbitrary spectrum may be generated by making use of the relation

$$T_n(\cos\theta) = \cos(n\theta)$$

where $T_n(\cdot)$ is the n^{th} order Chebychev polynomial (these may be generated with the *gen* program *chubby*). For example, if $f(x)$ is set equal to $a_0T_0(x) + a_1T_1(x) + \cdots + a_kT_k(x)$, then $f(\cos\theta)$ will have the spectral components $a_0\cos(0) + a_1\cos(\theta) + \cdots + a_k\cos(k\theta)$[22].

2.6.17. mult

General cmusic statement:

mult out[b] in[bnpv]* ;

The output is the product of all the inputs. Any number of inputs may be given.

---

22. For further information see "A Tutorial on Nonlinear Distortion or Waveshaping Synthesis" by C. Roads in *Foundations of Computer Music*, MIT Press, 1985.

2.6.18. neg

General cmusic statement:

```
neg out[b] in[bnpv] ;
```

The output is equal to minus the input.

2.6.19. nres

General cmusic statement:

```
nres out[b] in[bnpv] gain[bnpv] cf[bnpv] bw[bnpv]
   t1[dpv] t2[dpv] t3[dpv] t4[dpv] t5[dpv] t6[dpv]
   t7[dpv] t8[dpv] t9[dpv] t10[dpv] ;
```

cmusic.h macro definition:

```
#define NRES(out,in,gain,cf,bw) nres out in gain cf bw d d d d d d d d d d
```

nres[23] is a simple resonator which automatically adjusts its gain while bandwidth and center frequency change according to:

$$y[n] = gain*scale*(x[n] - r*x[n-2]) + c1*y[n-1] + c2*y[n-2]$$

where:

gain is an arbitrary gain factor,
cf (center frequency) is specified in units of Hz,
bw (bandwidth) is specified in units of Hz,
$r = e^{-\pi bw/R}$, where R is the sampling rate in Hz,
$scale = 1 - r$
$c1 = 2r\cos(2\pi \cdot cf/R)$, and
$c2 = -(r^2)$

Note that the gain at the peak of the resonance region will always be *gain*, so other frequencies may be severely attenuated if the bandwidth is small.

nres is discussed in the introduction to this manual.

2.6.20. osc

General cmusic statement:

```
osc out[b] amp[bnpv] incr[bnpv] table[fnpv] sum[bdnpv] ;
```

A truncating, table lookup oscillator. Dynamic variables are recommended for *sum* locations. *table* may be either explicit, or a number may be given. If the number is n, then $floor(n)$ is use as a wavetable index. Operation of the *osc* unit generator is discussed in the introduction to this manual.

2.6.21. out

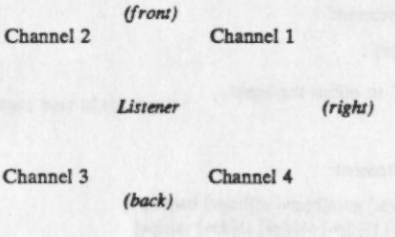
General cmusic statement:

```
out in[bdnpv]* ;
```

The number of inputs to the *out* unit generator *must* be equal to the number of sound channels. The first input is summed into output channel 1, the second input is summed into output channel 2, etc. The *set* command may be used to specify the number of channels.

When it is necessary, cmusic assumes the following correspondence between sound channels and loudspeakers placement:

23. *nres* is based on the Smith-Angell algorithm described in the *Computer Music Journal* Volume 6, Number 4, pp. 36–40.



2.6.22. rah

General cmusic statement:

```
rah out[b] amp[bnpv] incr[bnpv] pos[dpv] from[dpv] to[dpv] ;
```

rah chooses new random values at a rate of f Hz, where f is determined by the *incr* input. *rah* is the same as *ran* except that instead of moving in straight lines between random values, the random values are held steadily until a new one is chosen.

2.6.23. ran

General cmusic statement:

```
ran out[b] amp[bnpv] incr[bnpv] pos[dpv] from[dpv] to[dpv] ;
```

The output signal of *ran*[24] consists of straight line segments which travel from one random value to another. The frequency with which new random values are chosen is determined by *incr*.. A new random value is chosen every $L/incr$ samples, or at a rate of $R \cdot incr/L$ Hz, where L is the prevailing default function length and R is the sampling rate. Setting *incr* to 100Hz, for example, would cause a new random value to be chosen every 10 milliseconds. The random values lie in the range ±1, and the output signal is scaled by *amp*. The last three arguments are typically dynamic (*d*) variables, used for temporary storage by *ran*. The spectrum of the output signal is sinc-squared, symmetrically centered around 0 Hz with a bandwidth given by *increment*. It may by centered around an arbitrary center frequency by convolution with a sinusoid at that frequency.

The following score fragment produces bandpass noise centered around 1000 Hz with an approximate bandwidth from 900 to 1100 Hz:

```
ins 0 noise ;
   ran b1 p5 p7 d d d ;
   osc b1 b1 p6 f1 d ;
   out b1 ;
end ;
GEN2(f1) 1 1 ;
note 0 noise 4 0dB 1000Hz 200Hz ;
```

2.6.24. sah

General cmusic statement:

```
sah out[b] in[b] period[bnpv] temp1[dnpv] temp2[dnpv] ;
```

The *sah* unit generator looks at its input and holds it for *period* samples. For instance, for an input sequence x0, x1, x2, x3, etc., setting *period* to 1.0 will cause it to output x0, x0, x2, x2, x4, x4, etc. *period* is

24. For further information, see *The Technology of Computer Music* by Max V. Mathews with J. E. Miller, F. R. Moore, J.-C. Risset and J. R. Pierce, MIT Press, 1969, pp. 68–72.

the length of time to hold, in samples, and may be set to zero. A negative *period* is treated like a zero *period*. The "ms" postoperator (see *expressions*) may be used, for example, to specify the *period* in milliseconds.

Care must be exercised in the use of this sample and hold unit generator to avoid clicks due to discontinuities in the output sample stream. It may be desirable to "smooth" *sah* output with a lowpass filter (see *flt*, above).

2.6.25. seg

General cmusic statement:

seg out[b] amp[bnpv] table[fnpv] sum[dpv] incr[npv]* ;

seg is a special version of the oscillator specialized around the tasks of amplitude envelope generation. A wavetable is defined for use with *seg* so that it contains *N equal-length* segments. *seg* then scans these segments at variable rates over the duration of a note event, allowing one segment to control the shape of the attack, another to control the shape of the pseudo-steady state, another the initial decay, and so on. The *table* may be either a wavetable function (e.g., *f* 3), or a wavetable function index (e.g., 3).

The wavetable function is divided into *N* equal-length segments. *N* is then the number of *incr* fields given in the *seg* statement (typically 3, corresponding to attack, steady state, and decay times). *incr* fields with zero values may be used to signify that the corresponding segment duration is to be computed automatically from the duration that remains after all nonzero increments have been subtracted from the total note duration. For example, the following instrument plays with 100ms attack and decay times, with the steady state adjusted to fill up the note duration:

```
ins 0 env ;
   seg b1 p5 f2 d .1sec 0 .1sec ;
   osc b1 b1 p6 f1 d ;
   out b1 ;
end ;
GEN3(f2) 0 1 1 0 ; {a 3-segment trapezoidal envelope}
```

The following instrument allows attack and decay to be set directly, with steady state duration computed automatically:

```
ins 0 env ;
   seg b1 p5 f2 d p7 0 p8 ;
   osc b1 b1 p6 f1 d ;
   out b1 ;
end ;
GEN3(f2) 0 1 1 0 ;
```

More than one envelope segment may be computed automatically. In the next example, a 4-segment envelope would be adjusted over a 1-second note duration thus: segment 1 = .1 sec, segment 2 = .4 sec, segment 3 = .4 sec, segment 4 = .1 sec :

```
ins 0 env ;
   seg b1 p5 f2 d p7 0 0 p8 ; {p7 & p8 control att & dec times}
   osc b1 b1 p6 f1 d ;
   out b1 ;
end ;
GEN3(f2) 0 1 1 .5 0 ; {4 segment envelope}
SINE(f1) ;
note 0 env 1 0dB 440Hz .1sec .1sec ;
```

seg assumes that the sum of all nonzero *incr* fields will be strictly greater than the total duration *p4*. If this condition is not satisfied, the results may or may not be acceptable. In particular, if the sum of all nonzero increments is exactly equal to the total duration *p4*, a division by zero may result.

2.6.26. shape

General cmusic statement:

```
shape out[b] table[fnpv] sum[bdnpv] ;
```

shape is a truncating, table lookup oscillator with amplitude automatically set to 1, and frequency automatically set to *p4sec*. In other words, this unit generator simply scales the function given in *table* to the note duration. Dynamic variables are recommended for *sum* locations. *table* may be either explicit, or a number may be given. If the number is *n*, then *floor*(*n*) is used as a wavetable index. *shape* is equivalent to (but faster than) *osc* with amplitude set to 1. It is not significantly faster than the *freq* unit generator, but it obviates the need for a p-field set to *p4sec*.

2.6.27. show

General cmusic statement:

```
show in[bdfnpv]* ;
```

show causes the specified input(s) to be printed on the user's terminal, which is sometimes useful when debugging instrument definitions.

2.6.28. signum

General cmusic statement:

```
signum out[b] in[bnpv] ;
```

If the input is greater than or equal to zero, the output is set to 1.0, otherwise the output is set to –1.0.

2.6.29. sndfile

General cmusic statement:

```
sndfile out[b] amp[bnpv] incr[bnpv] filename[s] channel[npv] startframe[bnpv]
endframe[bnpv] pos[bdnpv] ptr[d] ;
```

sndfile reads in the specified *channel* of the file named by the *filename* string variable, starting at sample *startframe*, ending at sample *endframe* (if *endframe* = –1 then the end of the file is used). The specified increment (*incr*) is applied to the file (this is normally 1.0); if the end of the specified segment is reached, *sndfile* starts reading again from *startframe*.

The buffer size used by *sndfile* can be controlled with the "set sfbufsize = N ;" score statement (the default value for N is 4KB, and may cause swapping if many simultaneous read operations are used).

2.6.30. space

General cmusic statement:

```
space in[bnpv] nskip[npdv]
   t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d] t[d]
   [ x[bnpv] y[bnpv] theta[bnpv] len[bnpv] ]* ;
```

cmusic.h macro definitions:

```
#define SPACE(b,n) space b n d d d d d d d d d d d d

#define QUAD(In,Out)\
set quad ;\
set space = Out/2,Out/2 -Out/2,Out/2 -Out/2,-Out/2 Out/2,-Out/2 ;\
set room = In/2,In/2 -In/2,In/2 -In/2,-In/2 In/2,-In/2 ;\
set speakers = In/2,In/2 -In/2,In/2 -In/2,-In/2 In/2,-In/2 ;\
set revscale = .1 ;\
set t60 = (Out/10)^.5

#define STEREO(In,Out)\
set stereo ;\
set space = Out/2,Out/2 -Out/2,Out/2 -Out/2,-Out/2 Out/2,-Out/2 ;\
set room = In/2,In/2 -In/2,In/2 -In/2,-In/2 In/2,-In/2 ;\
set speakers = In/2,In/2 -In/2,In/2 ;\
set revscale = .1 ;\
set t60 = (Out/10)^.5
```

space is the cmusic spatialization unit generator[25]. The SPACE macro takes 2 arguments. The first argument is an i/o block (*b*) which is a source sound to be located in space at the coordinates *x*,*y*. The second argument is an obsolete parameter which should be set to 1. The set of parameters *x*, *y*, *theta*, *amp*, and *back* is called a *radiation vector*. Any number of radiation vectors may be given to cause the sound to seem to radiate from several locations at once, with each source having its own directional characteristics.

x and *y* are specified directly in meters—coordinates may lie in the range ±*Out* (see the STEREO and QUAD macro definitions, above). *theta* is a direction for a radiation vector (in radians, with 0 pointing to the right). *amp* is the length of the radiation vector (it simply scales the amplitude). *back* is the relative radiation amplitude in the direction opposite to *theta* (0 gives a very directional source, 1 gives an omnidirectional source, values in between give cardiod shapes in between—omnidirectional sources generally give better results). The relative amount of global reverb is specified with a *set* statement—the example below gives some good settings for everything.

25. For more information on the *space* unit generator, see "A General Model for Spatial Processing of Sounds" by F. R. Moore, *Computer Music Journal* Volume 7, Number 3, Fall, 1983.

```
#include <carl/cmusic.h>

STEREO(3,12) ;

ins 0 one ;
   seg  b4 p5 f4 d .1sec 0 .1sec ;
   osc  b5 b4 p10 f5 d ;
   osc  b2 p7 p8 f2 d ;
   osc  b3 p7 p9 f3 d ;
   adn  b3 b3 p11 ;
   osc  b1 b5 p6 f1 d ;
   SPACE(b1,1) b2 b3 0 1 0dB ;
end ;

SAWTOOTH(f1) ;
SINE(f2) ;
COS(f3) ;
GEN1(f4) 0,0 1/3,1 2/3,1 1,0 ;
PULSE(f5) ;
GEN3(f6) 3/4 -1/4 ;

{play 4 second note that moves in circular path centered
about point (0,20). Circle has radius of 10 meters. Sound
takes 2 seconds to complete one circular movement}

note 0 one 4 0dB 440Hz 10 2sec 2sec 11Hz 20 ;

sec ;

ter 4 ; {allow 4 seconds at end for global reverb to die}
```

2.6.31. splice

General cmusic statement:

```
splice out[b] amp[bnpv] incr[bnpv] filename[s]
   channel[npv] startframe[bnpv] endframe[bnpv]
   pos[bdnpv] ptr[d] old[dnpv] older[dnpv] oldest[dnpv] ;
```

splice reads in the specified channel of the soundfile named by the string variable, starting at a point which minimizes discontinuity with the three most recent values in old, older, and oldest. *startframe* and *endframe* are used to specify lower and upper bounds on the allowable startpoint. The specified increment is then used to move through the soundfile until the note is turned off or until the end of the file is reached, whichever comes first. This unit generator works just like *sndfile* except that:

1) *startframe* and *endframe* have different meanings,
2) there is no wrap-around at the end of the file, and
3) three extra dynamic variables are required.

A typical usage of *splice* is illustrated in the following score:

```
#define R 16K
var 0 v1 0 0 0 ;

ins 0 read ;
   splice b1 1 1 s1 1 p5 p6 d d v1 v2 v3 ;
   out b1 ;
end ;

var 0 s1 "soundfile1" ;
note 0 read .3 0 0 ;
var p2+p4 s1 "soundfile2" ;
note p2 read .7 .61*R p5+200 ;
```

The *var* statement defines the three variables *v*1, *v*2, *v*3; these are essential if *splice* is to function properly. Furthermore, a separate set of variables is required for each channel. Thus, if the above score were to work for stereo, it would have to be changed to read:

```
set stereo ;
#define R 16K
var 0 v1 0 0 0 0 0 0 ;

ins 0 left ;
   splice b1 1 1 s1 1 p5 p6 d d v1 v2 v3 ;
   out 0 b1 ;
end ;

ins 0 right ;
   splice b2 1 1 s1 2 p5 p6 d d v4 v5 v6 ;
   out b2 0 ;
end ;

var 0 s1 "soundfile1" ;
note 0 left .3 0 0 ; note 0 right .3 0 0 ;
var p2+p4 s1 "soundfile2" ;
note p2 left .7 .61*R p5+200 ; note p2 right .7 .61*FSR p5+200 ;
```

If *endframe* = *startframe* (or if *endframe* = –1), then *splice* is equivalent to *sndfile* with *endframe* = –1 (except for the lack of wrap-around). If *endframe* > *startframe*, then the intervening frames are evaluated to determine the best startpoint. If possible, it is suggested that *endframe* be chosen so that these intervening frames contain several complete cycles of the signal. But remember that both *startframe* and *endframe* must be specified in terms of samples as opposed to seconds.

The best startpoint is computed as follows:

1) a Taylor series expansion is used to compute the next two points succeeding oldest, older, and old, and
2) each successive pair of points between *startframe* and *endframe* is compared to these two computed points, and the pair which differs by the least is chosen as the startpoint.

It is important to note that the specified amplitude value is used both in computing old, older, and oldest, and in picking the new startpoint. Thus, different amplitude values can be used in successive calls to *splice*, but only with care.

splice should be useful for butting together two soundfiles (e.g. when a long signal processing job has died and been restarted in the middle), and also for removing isolated clicks.

2.6.32. sqroot

General cmusic statement:

```
sqroot out[b] in[bnpv] ;
```

The output of *sqroot* is equal to the square root of its input. Naturally, the input had better be greater than zero.

2.6.33. square

General cmusic statement:

```
square out[b] in[bnpv] ;
```

The output of the *square* is equal to the square of its input.

2.6.34. trans

General cmusic statement:

```
trans out[b] no[dpv] len[dpv] i[dpv] [ t[npv] v[npv] a[npv] ]* t[npv] v[npv] ;
```

trans is the unit generator equivalent of *gen4*, which is a general exponential transition generator. The three temporary variables, *t*1, *t*2, and *t*3, are normally *d*'s. The rest of the line gives a description of a transition path exactly like a *gen4* definition, except that the transition path occurs over the duration of a note event. Any number of time-value (*t-v*) pairs may be given, and the transition parameters (*a*'s) work as they do in *gen4*: 0 yields a straightline transition, negative numbers yield exponential transitions, and positive numbers yield inverse exponential transitions.

The power of this generator lies in the fact that the transition points and parameters may be connected to p-fields, allowing such things as easy glissando specification, loudness contouring, etc. For example, the following score plays a 1-second glissando from 440Hz to 880Hz, followed by a 2-second gliss to 100Hz:

```
ins 0 gliss ;
   trans b1 d d d 0,p6 0 1/3,p7 0 1,p8 ;
   osc   b1 p5 b1 f1 d ;
   out   b1 ;
end ;
SINE(f1) ;
note 0 gliss 3 -6dB 440Hz 880Hz 100Hz ;
ter ;
```

Naturally, *trans* runs considerably slower for nonzero transition parameter values than it does for straightline transitions. Also, *trans* should not be used to generate the same control function over and over—that is what wavetables are for.

2.6.35. version

General cmusic statement:

```
version out[b] amp[bnpv] incr[bnpv] filename[s] channel[npv] a[npv]
   b[npv] c[npv] d[npv] tau[npv] Rorig[npv] Forig[npv]
   ipos[dnpv] fpos[dnpv] lasto[dnpv] ptr[dnpv] sum[dnpv] ;
```

version is a unit generator that creates a new version of a sound which is stored on a soundfile. The new version may have a new pitch or duration or both. Thus a digitally recorded note from an acoustic instrument may be retuned to a new pitch, and/or lengthened or shortened. The output is scaled by *amp*, allowing new amplitudes, amplitude modulation, or new envelopes to be superimposed. The *incr* value specifies the frequency to be generated (e.g., "440Hz"), the *Forig* value specifies the frequency of the original file as an increment (e.g., "220Hz"). The original sampling rate is given by the *Rorig* argument.

As in the *sndfile* unit generator, the *filename* argument must be a string variable which has been defined with the name of the soundfile to be accessed, and the *channel* number selects which out of several possible channels on the file is to be used.

The *a*, *b*, *c*, and *d* values specify the sample numbers of

- the start of the note (on the file),
- the start of the (possibly repeated) steady state,
- the end of the (possibly repeated) steady state, and
- the end of the note, respectively.

The *tau* value specifies the number of samples to be used in a cross-faded transition between the end of one steady state and the beginning of the next, or between the first part of the note and the decay section (which starts at sample *c*). A good value for *tau* will probably be the number of samples in a pitch period of the original sound.

The last 5 arguments (there are 15 in all) will normally be *d*'s. It is an error to try to play a new version of a note which has fewer samples in it than the decay portion of the original (*d*–*c* samples).

2.6.36. zdelay

General cmusic statement:

```
zdelay out[b] in[bnpv] maxtime[npv] table[dnpv] tlen[dnpv]
pos[dnpv] gain[bnpv] delay[bnpv]* ;
```

zdelay output is equal to its input delayed by *delay* seconds and scaled by *gain*. An arbitrary number of gain/delay taps may be given—the output is the sum of all tap outputs. *zdelay* is, therefore, an arbitrary FIR (i.e., tapped delay-line) filter with continuously adjustable taps.

Since the amount of delay is continuously and dynamically variable, *zdelay* is useful for effects such as pitch shifting, flanging, etc. The maximum allowable delay is set by the user in *maxtime* (also given in seconds). It is an error for any delay time to exceed the value given in *maxtime* for any given note (negative delays are also illegal). *table*, *tlength*, and *pos* arguments are typically dynamic variables. Note that while *gain* and *delay* values may be dynamic signals or constants (*b*, *v*, *p*, or *n*), *maxtime must* be a constant (*v*, *p*, or *n*).

3. Advanced Topics

cmusic is a complex program, and uses for it can be even more complex. making it impossible to discuss cmusic applications in any comprehensive way. This section includes some suggestions, hints, kinks, and a little speculation about how cmusic may be used.

3.1. Documentation

First of all, this document is probably obsolete even as you read it. cmusic is a research program, which means that its properties vary from time to time, new commands and unit generators are implemented, old bugs get fixed, and new techniques are discovered. Where this document and the online documentation differ, the online documentation will nearly always be correct.

This implies that it will be extremely helpful for the reader to become as adroit as possible in understanding how to use the online documentation—not only for cmusic—but for all CARL software.

The major sources of information aside from the *CARL Startup Kit* are the *help* and *man* commands. The difference between *help* "files" and *man* "pages" is largely academic—originally the idea of a *man* page was that UNIX programs were to be so simple that they would require just one reference page for their full explanation to an advanced user. Needless to say, many programs are so complicated as to make this a prime example of wishful thinking.

Nevertheless, *man* "pages" are still considered to be *definitive* documents about the operation of any program they describe. They are not, however, intended to be *tutorial*—they are references only, and for advanced users at that!

help "files" are intended to provide extensions to the *man* pages for users needing more "help": they often include tutorial examples, for instance.

Online documentation takes some getting used to, and many people find it a bit difficult to follow, since a terminal screen is not so pleasing to the eye as good quality hardcopy. Nevertheless, things change too quickly to allow revisions of hardcopy documents for each and every change in programs, let alone operating systems.

3.2. Debugging cmusic Scores

Two useful debugging tools built into cmusic are the *verbose* and *list* file options, both of which produce a blow-by-blow description of what cmusic is up to at any given moment.

The *verbose* mode can be used with short scores (it can be turned on either with the "set verbose ;" statement or via the –v flag when cmusic is run). For longer runs, however, the "set list ;" option is often indispensable.

A useful technique is to create a ".list" file with the same first name as the score file by including the "set list ;" statement at the head of a score—typically right after the "#include <carl/cmusic.h>" statement. The UNIX "tail" command may be used at any time to examine the last few lines of the ".list" file, and the "tail –f" command option allows the user to attach the terminal to a growing ".list" file in a revokable manner.

If things seem to be going poorly—for example, no sound comes out—it is often helpful to examine the signals passing among the unit generators in the instruments being played. This can be done by "commenting out" certain of the original unit generator statements (by enclosing them in curly braces) and attaching the output of a questionable signal directly to the *out* unit generator. The signal will then pour onto the user's terminal screen if cmusic is run with its standard output connected to the terminal.

Observation of such signals is often difficult because there are just so many numbers that their pattern can be difficult to see. One technique here is to reduce the sampling rate temporarily (use the –R flag), and/or to attach the cmusic output to a graphics program such as "graph." If cmusic output is connected to a UNIX pipe, cmusic will produce binary floatsams, and "graph" requires ASCII input. A CARL utility program called "btoa" is provided specifically for insertion into a pipe-line where sample data must be converted from binary to ASCII. For example, the commands

```
cmusic score.sc
```

and

```
cmusic score.sc | btoa
```

produce exactly the same list of numbers on the user's terminal.

The technique of "commenting out" score statements applies equally well to note lists. Often it is desirable to run just a portion of a score for testing purposes—curly braces may be used to temporarily turn off a part of the score (but be careful about action times, or you may accidentally specify that 100 seconds of silence are to be generated before any note will sound!).

Clicks are a perennial problem in digital sound synthesis. Clicks are caused by discontinuities in the waveform of the digital signal. Such discontinuities can arise from many sources.

Turning on or shutting off a note with a nonzero amplitude will almost always create a click. Care must be exercised to ensure that envelope functions really begin and end with 0.0, and that they don't rise (or fall) too rapidly from this value of infinite repose. Even an overly-enthusiastic attack time can create a click sometimes and not others, depending on where the peaks of the enveloped waveform happen to fall under the control curve.

Clicks may suddenly appear when a sound is spatialized with the *space* unit generator, even if no clicks were evident in the original sound, due to the click-enhancing effect of reverberation. Discontinuities in sound paths are translated directly into amplitude discontinuities—as when the sound source suddenly goes from here to there without going in between—often making it necessary to use *iosc* instead of *osc* unit generators for soundpath control.

Any time the magnitude of the total amplitude exceeds 1.0, cmusic is forced to "clip" offending samples in a way reminiscent of hard analog limiting—so hard, in fact, that the sound is easily disfigured by it.

Even a sudden change of a non-amplitude parameter—such as frequency—can produce clicks under certain circumstances, depending on the abruptness of the waveform change.

3.3. Oscillator Phase Connection

The oscillator unit generators all use sum locations that are typically attached to dynamic state variables (*d*'s). Such variables are allocated at the beginning of a note as needed, and their values are initialized to 0.0. Thus, two contiguous, non-enveloped notes in a row on an *osc* unit generator are likely to produce a click because the oscillator phase will be reset at the beginning of each note, as shown in the following score example.

```
#include <carl/cmusic.h>
ins 0 toot ;
   osc b1 p5 p6 f1 d ;
   out b1 ;
end ;
SINE(f1) ;
note 0 toot 1 -6dB 100Hz ;
note 1 toot 1 -6dB 110Hz ;
ter ;
```

A useful technique for "connecting" the phase of the two notes is to use a global variable (*v*) for the oscillator sum location—this memory location is allocated *permanently* (for the duration of the run), so the phase (i.e., sum-of-increments) value at the end of the first note will be remembered at the beginning of the second, as in the following example:

```
#include <carl/cmusic.h>
ins 0 toot ;
   osc b1 p5 p6 f1 v1 ;
   out b1 ;
end ;
SINE(f1) ;
note 0 toot 1 -6dB 100Hz ;
note 1 toot 1 -6dB 110Hz ;
ter ;
```

The beginning of the second note in the last example is very likely *not* to produce a click because the change in frequency between the two notes is not too great. Note, however, that it is now impossible to play more than one note at a time on instrument *toot*, since two simultaneous notes would compete destructively for the contents of global variable *v*1.

3.4. Global Control

cmusic operates by observing the action time of the command at hand in the score, starting from the beginning, of course. Once an action time is encountered that is greater than the "current" action time (the greatest time to which sound has been synthesized), cmusic starts generating samples. Sample generation continues until "something happens", which means that either a command is to be executed, or a note terminates, in which case resources for that note must be deallocated.

cmusic does not produce one sample at a time but runs in *blocks* of samples—the length of a sample synthesis block is equal to the length of the i/o blocks (*b*'s).

For example, if it is time to start synthesizing a note, cmusic looks up the definition for the relevant instrument and calls the unit generator programs one at a time in the order they are stated in the instrument definition. Information is passed to the individual unit generators about where to find information about parameter information—a *b* will be a pointer to an array (i/o block), a *p* will point to a single value within the list of note parameters for the current note, and so on. Each unit generator is instructed to generate either *B* samples (where *B* is the number of samples in an i/o block), or fewer, if the note is to terminate before *B* samples have gone by from the current synthesis time.

This manner of operation allows an interesting form of global control to be used by the cmusic score-writer. A "dummy" instrument may be defined that outputs into an i/o block, let's say a *shape* unit generator outputs into *b*10, as in the following example:

```
ins 0 cresc ;
   shape b10 f2 d ;
end ;
GEN4(f2) 0,-30dB -2 1,-20dB ;
```

Notice that there is no *out* unit generator included in the definition of instrument *cresc*. The values in *b*10 seem to be going nowhere!

Other instruments, however, may refer to *b*10 as part of their definitions. In this case, since the *b*10 values will gradually change from –30 to –20 dB over the course of any note played on instrument *cresc*, we may deduce that it is designed for longterm amplitude control. In fact, *cresc* will produce a *crescendo* when used in conjunction with other instruments in the following manner:

```
ins 0 cresc ;
   shape b10 f2 d ;
end ;
GEN4(f2) 0,-30dB -2 1,-20dB ;
ins 0 toot ;
   seg b1 b10 f3 d p6 0 p7 ;
   osc b1 b1 p5 f1 d ;
   out b1 ;
end ;
GEN4(f3) 0,0 -3 1/3,1 0 2/3,.1 -2 1,0 ;
SAWTOOTH(f1) ;

note 0 cresc 5 ; {crescendo over 5 seconds}

note 0 toot 1 A(0) .1sec .1sec ;
note 0 toot 1 Cs(1) .1sec .1sec ;

note 2 toot 1.5 B(0) .05sec .1sec ;
note 2.3 toot 2 Ds(1) .05sec .1sec ;

note 3.6 toot 1.4 E(1) .01sec .1sec ;

sec ;
```

Several advanced techniques are demonstrated in this example.

First, *b*10 is used to control the amplitude of another instrument, which plays chords with various pitches and attack times. *b*10 is simply fed into the *amp* input of the *seg* unit generator controlling the peak note amplitude—the amplitude envelopes are continuously being fed a greater and greater peak value as the first five seconds of the score go by.

Second, *b*1 is used both as an input and an output of the *osc* unit generator. This is perfectly legal since the information in *b*1 is examined before the output is produced—we only have to be careful not to use the same i/o block to hold more than one thing at one time! No unit generator may use *b*10 while the *cresc* unit generator is operating, for example.

This demonstrates that even though i/o blocks are dynamic they are allocated at the beginning of any note that uses them, and remain usable as long as that note event is active. Their contents will always be whatever was last put there from any source.

Such global control techniques may be used for more than the production of *crescendi*, of course. Any and all instrument parameters may be modulated in this way, yielding an entire layer of control possibilities independent of the note event level.

3.5. Advanced Uses of sndfile

The sndfile unit generator allows cmusic to be used as a general mixing program for digital recordings and soundfiles that result from other synthesis runs. Global control functions may be used to specify long term mixing levels, filtering (equalization) parameters, panning, spatialization, and so on.

In addition, the sndfile unit generator can be used to read in the results of sound analysis programs such as the *phase vocoder* (*pvoc*). pvoc analysis output may be viewed as the output of a large number of special bandpass filters, where each pvoc filter describes the time varying amplitude and frequency of an individual sinusoidal component that lies within its bandpass region.

With the proper flag specifications (see the pvoc documentation) pvoc analysis results may be stored in a special soundfile. The format of this special soundfile is $N+2$ channels ($N/2+1$ amplitude channels and $N/2+1$ frequency channels all interleaved). Odd channels are time varying amplitudes and even channels are time varying frequencies (channels are numbered 1 through N+2). The first two channels contain the

amplitude and frequency of the DC (zero frequency) filter, so the data for the fundamental is usually in channels 3 (amplitude) and 4 (frequency).

The general additive synthesis shown in Figure 14 may be replaced by a simple oscillator with a sndfile unit generator controlling its amplitude and another controlling its frequency for each of the $N+2$ analysis channels. Only one such cmusic instrument needs to be defined, of course, since it is possible to effect the resynthesis by playing $N+2$ simultaneous notes on it, each contributing a single additive component to the resulting sound.

Needless to say, the special soundfile that holds the analysis results can be quite large. Its size is determined by the number of pvoc channels (N), and the amount of window overlap used in the analysis.

4. APPENDIX I: The cmusic Command

cmusic [-o] [[-v] [-n] [-t] [-q] [-Rx] [-Lx] [-Bx] [score_file] > output

cmusic flags given on the command line override options in the score_file.

-o tells cmusic to produce no sample output (debug mode)

-v sets the verbose option; -v- turns it off (default = off).

-n sets the notify option; -n- turns it off (default = off).

-t sets the timer option; -t- turns it off (default = off).

-q turns off any verbose, timer, or notify options set elsewhere.

-Rx sets the sampling rate to x (default = 16K).

-Lx sets the default function length to x (default default = 1K).

-Bx sets the io block length to x (default = 256).

Flag symbols must not be combined, i.e., "-tn" will not work, but "-t -n" will.

If no score_file is given, cmusic reads its standard input.

If its standard output is connected to a terminal, cmusic generates ASCII sample values on the screen; if the standard output is not a terminal, binary values (floats by default) are produced.

Detected score errors generally cause sample synthesis to stop—the remainder of the score is scanned for further errors, if possible.

6. APPENDIX II: Sample Score Files

A few briefly annotated score examples are included here for reference. These scores are intended only to give a flavor the the idiomatic use of the cmusic score language, and are not compositional models.

The following score synthesizes several notes with just pitch and just and tempered timbres.

```
#include <carl/cmusic.h>
set list ;
set funclength = 8K ;
set srate = 48K ;
ins 0 comp ;
{envelope}      osc  b1 p5 p7 f2 d ;
{carrier}       osc  b1 b1 p6 f1 d ;
{output}        out  b1 ;
end ;

GEN2(f1) 1 1 ;
GEN4(f2) 0,0 -2 .2,1 -1 .5,5 -2 1,0 ;

#define AMP (-18dB)

#define TEMP(time, dur, amp, pitch)\
note time comp dur amp pitch P ;\
note time comp dur amp/2 pitch*2 P ;\
note time comp dur amp/3 pitch*2*2^(7/12) P ;\
note time comp dur amp/4 pitch*4 P ;\
note time comp dur amp/5 pitch*4*2^(4/12) P

#define JUST(time, dur, amp, pitch)\
note time comp dur amp pitch P ;\
note time comp dur amp/2 pitch*2 P ;\
note time comp dur amp/3 pitch*3 P ;\
note time comp dur amp/4 pitch*4 P ;\
note time comp dur amp/5 pitch*5 P

{
JUST(0,2,(-30dB),(C(0))) ;
sec ;
sec .5 ;
TEMP(0,2,(-30dB),(C(0))) ;
sec ;
sec .5 ;
}

JUST(0,1.5,(-30dB),(P1(-2)*C(0))) ;      {(C(-2))} ;
JUST(0,1.5,(-30dB),(P1(-1)*C(0))) ;      {(C(-1))} ;
JUST(0,1.5,(-30dB),(P5(-1)*C(0))) ;      {(G(-1))} ;
JUST(0,1.5,(-30dB),(P1(0)*C(0))) ;       {(C(0))} ;
JUST(0,1.5,(-30dB),(M3(0)*C(0))) ;       {(E(0))} ;
JUST(0,1.5,(-30dB),(P5(0)*C(0))) ;       {(G(0))} ;

JUST(2,8,(-24dB),(P1(-2)*P4(0)*C(0))) ; {(F(-2))} ;
JUST(2,8,(-24dB),(P5(-2)*P4(0)*C(0))) ; {(C(-1))} ;
JUST(2,8,(-24dB),(M3(-1)*P4(0)*C(0))) ; {(A(-1))} ;
```

```
JUST(2,8,(-24dB),(P5(-1)*P4(0)*C(0))) ; {(C(0)) ;}
JUST(2,8,(-24dB),(M7(-1)*P4(0)*C(0))) ; {(E(0)) ;}
JUST(2,8,(-24dB),(M2(0)*P4(0)*C(0))) ; {(G(0)) ;}

JUST(10,1.5,(-24dB),(P1(-3)*P4(0)*C(0))) ; {(F(-3)) ;}
JUST(10,1.5,(-24dB),(P1(-2)*P4(0)*C(0))) ; {(F(-2)) ;}
JUST(10,1.5,(-24dB),(M3(-1)*P4(0)*C(0))) ; {(A(-1)) ;}
JUST(10,1.5,(-24dB),(P5(-1)*P4(0)*C(0))) ; {(C(0)) ;}
JUST(10,1.5,(-24dB),(m7(-1)*P4(0)*C(0))) ; {(Ef(0)) ;}
JUST(10,1.5,(-24dB),(M2(0)*P4(0)*C(0))) ; {(G(0)) ;}

JUST(12,8,(-24dB),(P1(-3)*C(0))) ;      {(C(-3)) ;}
JUST(12,8,(-24dB),(P5(-2)*C(0))) ;      {(G(-2)) ;}
JUST(12,8,(-24dB),(P1(0)*C(0))) ;       {(C(0)) ;}
JUST(12,8,(-24dB),(M3(0)*C(0))) ;       {(E(0)) ;}
JUST(12,8,(-24dB),(P5(0)*C(0))) ;       {(G(0)) ;}
JUST(12,8,(-24dB),(M7(0)*C(0))) ;       {(B(0)) ;}

sec ;
sec 2 ;

TEMP(0,1.5,(-30dB),(P1(-2)*C(0))) ;     {(C(-2)) ;}
TEMP(0,1.5,(-30dB),(P1(-1)*C(0))) ;     {(C(-1)) ;}
TEMP(0,1.5,(-30dB),(P5(-1)*C(0))) ;     {(G(-1)) ;}
TEMP(0,1.5,(-30dB),(P1(0)*C(0))) ;      {(C(0)) ;}
TEMP(0,1.5,(-30dB),(M3(0)*C(0))) ;      {(E(0)) ;}
TEMP(0,1.5,(-30dB),(P5(0)*C(0))) ;      {(G(0)) ;}

TEMP(2,8,(-24dB),(P1(-2)*P4(0)*C(0))) ;  {(F(-2)) ;}
TEMP(2,8,(-24dB),(P5(-2)*P4(0)*C(0))) ;  {(C(-1)) ;}
TEMP(2,8,(-24dB),(M3(-1)*P4(0)*C(0))) ;  {(A(-1)) ;}
TEMP(2,8,(-24dB),(P5(-1)*P4(0)*C(0))) ;  {(C(0)) ;}
TEMP(2,8,(-24dB),(M7(-1)*P4(0)*C(0))) ;  {(E(0)) ;}
TEMP(2,8,(-24dB),(M2(0)*P4(0)*C(0))) ;   {(G(0)) ;}

TEMP(10,1.5,(-24dB),(P1(-3)*P4(0)*C(0))) ; {(F(-3)) ;}
TEMP(10,1.5,(-24dB),(P1(-2)*P4(0)*C(0))) ; {(F(-2)) ;}
TEMP(10,1.5,(-24dB),(M3(-1)*P4(0)*C(0))) ; {(A(-1)) ;}
TEMP(10,1.5,(-24dB),(P5(-1)*P4(0)*C(0))) ; {(C(0)) ;}
TEMP(10,1.5,(-24dB),(m7(-1)*P4(0)*C(0))) ; {(Ef(0)) ;}
TEMP(10,1.5,(-24dB),(M2(0)*P4(0)*C(0))) ; {(G(0)) ;}

TEMP(12,8,(-24dB),(P1(-3)*C(0))) ;      {(C(-3)) ;}
TEMP(12,8,(-24dB),(P5(-2)*C(0))) ;      {(G(-2)) ;}
TEMP(12,8,(-24dB),(P1(0)*C(0))) ;       {(C(0)) ;}
TEMP(12,8,(-24dB),(M3(0)*C(0))) ;       {(E(0)) ;}
TEMP(12,8,(-24dB),(P5(0)*C(0))) ;       {(G(0)) ;}
TEMP(12,8,(-24dB),(M7(0)*C(0))) ;       {(B(0)) ;}

ter ;
```

The following score defines a frequency-modulated instrument and plays a note on it.

```
#include <carl/cmusic.h>
set funclength = 8K ;
set list ;

ins 0 fm ; {p5:p4sec, p6:peak amp, p7:fc}
   {amp env}   osc b1 p6 p5 f2 d ;
   {df env}    osc b2 1 p5 f3 d ;
   {fm env}    osc b3 1 p5 f4 d ;
   {mod osc}   osc b5 b2 b3 f1 d ;
   {sum}       adn b6 b5 p7 ;
   {carrier}   osc b7 b1 b6 f1 d ;
   {output}    out b7 ;
end ;
{f1 – sine}
SINE(f1) ;
{f2 – amp env}
GEN4(f2) 0,0 –1 .05,1 0 .95,1 –2 1,0 ;
{f3 – df env}
GEN4(f3) 0,0 0 1/8,6Hz 0 2/8,12Hz 0 3/8,300Hz 0 4/8,900Hz 0
         7/8,900Hz 0 8/8,0 ;
{f4 – fm env}
GEN4(f4) 0,1.5Hz (ln(2)) 4/8,3Hz (ln(100)) 7/8,300Hz 0 1,300Hz ;

note 0 fm 32 p4sec –12dB 300Hz ;

ter ;
```

The following score exploits cmusic macros in order to create a score shorthand similar to music notation.

```
#include <carl/cmusic.h>

set notify ;
set func = 4K ;

ins 0 t ;
   seg b2 1 f2 d 0 ;
   mult b5 b2 p5 ;
   mult b3 b2 p8 ;
   osc b4 b3 p7 f1 d ;
   adn b4 b4 p6 ;
   osc b1 b5 b4 f1 d ;
   out b1 ;
end ;
{p5–amp ; p6–Fc ; p7–Fm ; p8–dF ;}
ins 0 o ;
   seg b2 1 f4 d 0 ;
   mult b5 b2 p5 ;
   mult b3 b2 p8 ;
   osc b4 b3 p7 f1 d ;
   adn b4 b4 p6 ;
   osc b1 b5 b4 f1 d ;
   out b1 ;
end ;
ins 0 b ;
   seg b2 1 f4 d 0 ;
   mult b5 b2 p5 ;
   mult b3 b2 p8 ;
   osc b4 b3 p7 f1 d ;
   adn b4 b4 p6 ;
   osc b1 b5 b4 f1 d ;
   out b1 ;
end ;
SINE(f1) ;
GEN4(f2) 0,0 –2 .1,1 0 .8,.5 –2 1,0 ;
GEN5(f3) 1,1,0 2,1,0 3,1/2,0 4,1,0 5,1/3,0 6,1/4,0 7,1/5,0 ;
NORM(f3) ;
GEN4(f4) 0,0 –3 .1,1 0 .9,1 0 1,0 ;

#define amp (1/10)
#define T (4*(180MM))
#define STAC .6
#define NON .9
#define LEG 1.1
#define INDEX 7
#define HARM 1
{Macro to play a note AFTER the last one in the score}
#define NEXT(ins,pitch,dur,duty)\
note p2+p4 ins T*duty/(dur) amp   pitch  HARM*p6  INDEX*p7 ;\
note p2 ins T*duty/(dur) amp   pitch/2  HARM*p6  INDEX*p7 ;\
var  p2  p4  T/(dur)
```

```
{Macro to play a note WITH the last one in the score}
#define SAME(ins,pitch,dur,duty)\
var  p2    v1 p4 ;\
note p2    ins T*duty/(dur) amp   pitch HARM*p6  INDEX*p7 ;\
note p2    ins T*duty/(dur) amp   pitch/2 HARM*p6  INDEX*p7 ;\
var  p2    p4 v1

{Chord Macros}
#define CH2(p1,p2,ins,dur,duty)\
 NEXT(ins,p1,dur,duty) ; SAME(ins,p2-.5,dur,duty)

#define CHS2(p1,p2,ins,dur,duty)\
 SAME(ins,p1,dur,duty) ; SAME(ins,p2-.5,dur,duty)

#define CH3(p1,p2,p3,ins,dur,duty)\
 NEXT(ins,p1,dur,duty) ; SAME(ins,p2-.5,dur,duty) ; SAME(ins,p3-.7,dur,duty)

#define CHS3(p1,p2,p3,ins,dur,duty)\
 SAME(ins,p1,dur,duty) ; SAME(ins,p2-.5,dur,duty) ; SAME(ins,p3-.7,dur,duty)

#define CH4(p1,p2,p3,p4,ins,dur,duty)\
 NEXT(ins,p1,dur,duty) ;SAME(ins,p2,dur,duty) ;\
  SAME(ins,p3,dur,duty) ;SAME(ins,p4,dur,duty)

#define CHS4(p1,p2,p3,p4,ins,dur,duty)\
 SAME(ins,p1,dur,duty) ;SAME(ins,p2,dur,duty) ;\
  SAME(ins,p3,dur,duty) ;SAME(ins,p4,dur,duty)

{Notelist begins here – tune is by Mendelssohn}

    CH3(C(0),C(0),C(0),t,4*3,STAC) ;
     CH3(C(0),C(0),C(0),t,4*3,STAC) ;
      CH3(C(0),C(0),C(0),t,4*3,STAC) ;
CH3(C(0),C(0),C(0),t,4/3,NON) ;

 CH3(C(0),C(0),C(0),t,4*3,STAC) ;
  CH3(C(0),C(0),C(0),t,4*3,STAC) ;
   CH3(C(0),C(0),C(0),t,4*3,STAC) ;
CH3(C(0),C(0),C(0),t,4/3,NON) ;

 CH3(C(0),C(0),C(0),t,4*3,STAC) ;
  CH3(C(0),C(0),C(0),t,4*3,STAC) ;
   CH3(C(0),C(0),C(0),t,4*3,STAC) ;
CH3(C(0),E(0),E(0),t,4,STAC) ;

 CH3(C(0),E(0),E(0),t,4*3,STAC) ;
  CH3(C(0),E(0),E(0),t,4*3,STAC) ;
   CH3(C(0),E(0),E(0),t,4*3,STAC) ;
    CH3(C(0),E(0),E(0),t,4,STAC) ;

    CH3(C(0),E(0),E(0),t,4*3,STAC) ;
     CH3(C(0),E(0),E(0),t,4*3,STAC) ;
      CH3(C(0),E(0),E(0),t,4*3,STAC) ;
CH3(C(0),E(0),G(0),t,4,STAC) ;
```

```
CH3(C(0),E(0),G(0),t,4*3,STAC) ;
 CH3(C(0),E(0),G(0),t,4*3,STAC) ;
  CH3(C(0),E(0),G(0),t,4*3,STAC) ;
   CH3(C(0),E(0),G(0),t,4,STAC) ;

    CH3(C(0),E(0),G(0),t,4*3,STAC) ;
     CH3(C(0),E(0),G(0),t,4*3,STAC) ;
      CH3(C(0),E(0),G(0),t,4*3,STAC) ;
CH4(C(1),E(1),Fs(1),C(2),o,2,LEG) ;
            CHS3(E(0),Fs(0),C(1),t,2,STAC) ;CHS2(A(-3),A(-2),b,2,LEG) ;
 CH4(B(0),Ds(1),Fs(1),B(1),o,16/7,LEG) ; CHS2(B(-3),B(-2),b,2,LEG) ;
  CH4(B(0),Ds(1),Fs(1),Fs(1),o,16,NON) ;
CH4(B(0),Ds(1),Fs(1),A(1),o,4,LEG) ; CHS2(E(-3),E(-2),b,2,LEG) ;
 CH4(B(0),E(1),G(1),G(1),o,4,LEG) ;
  CH4(A(0),D(1),F(1),F(1),o,4,NON) ; CHS2(F(-3),F(-2),b,2,LEG) ;
   CH4(F(0),A(0),D(1),D(1),o,4,NON) ;
CH4(E(0),G(0),C(1),C(1),o,2*9,LEG) ; CHS2(G(-3),G(-2),b,2,LEG) ;
 CH4(F(0),F(0),D(1),D(1),o,2*9,NON) ;
  CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
   CH4(F(0),F(0),D(1),D(1),o,2*9,NON) ;
    CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
     CH4(F(0),F(0),D(1),D(1),o,2*9,NON) ;
      CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
       CH4(D(0),D(0),B(0),B(0),o,2*9,NON) ;
        CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
 CH4(F(0),G(0),B(0),D(1),o,4,STAC) ; CHS2(G(-3),G(-2),b,2,LEG) ;
  CH4(F(0),G(0),G(0),G(0),o,16/3,STAC) ;
   CH4(F(0),G(0),B(0),D(1),o,16,NON) ;
CH4(E(0),G(0),C(1),E(1),o,4,LEG) ; CHS2(C(-3),C(-2),b,2,LEG) ;
 CH3(C(0),C(0),C(0),t,8,NON) ;
  CH3(E(0),E(0),E(0),t,8,NON) ;
   CH3(G(0),G(0),G(0),t,8,NON) ;
    CH3(C(1),C(1),C(1),t,8,NON) ;
     CH3(C(1),E(1),E(1),t,8,NON) ;
      CH3(C(1),E(1),G(1),t,8,NON) ;
CH2(A(-3),A(-2),b,4,LEG) ; CHS4(C(1),E(1),Fs(1),C(2),o,2,LEG) ;
          CHS3(E(1),Fs(1),C(2),t,2,STAC) ;
 CH2(Fs(-3),Fs(-2),b,4,LEG) ;
  CH2(B(-3),B(-2),b,4,LEG) ; CHS4(B(0),Ds(1),Fs(1),B(1),o,2,LEG) ;
   CH2(Ds(-3),Ds(-2),b,16/3,LEG) ;
    CH4(B(0),Ds(1),Fs(1),Fs(1),o,16,NON) ;
CH4(B(0),Ds(1),Fs(1),A(1),o,4,LEG) ; CHS2(E(-3),E(-2),b,4,LEG) ;
 CH4(B(0),E(1),G(1),G(1),o,4,LEG) ; CHS2(G(-3),G(-2),b,4,LEG) ;
  CH4(A(0),D(1),F(1),F(1),o,4,NON) ; CHS2(F(-3),F(-2),b,4,LEG) ;
   CH4(F(0),A(0),D(1),D(1),o,4,NON) ; CHS2(A(-3),A(-2),b,4,LEG) ;
CH4(E(0),G(0),C(1),C(1),o,2*9,LEG) ; CHS2(G(-3),G(-2),b,2,LEG) ;
 CH4(F(0),F(0),D(1),D(1),o,2*9,NON) ;
  CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
   CH4(F(0),F(0),D(1),D(1),o,2*9,NON) ;
    CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
     CH4(F(0),F(0),D(1),D(1),o,2*9,NON) ;
      CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
       CH4(D(0),D(0),B(0),B(0),o,2*9,NON) ;
        CH4(E(0),E(0),C(1),C(1),o,2*9,NON) ;
```

```
 CH4(E(0),G(0),C(1),E(1),o,4,NON) ; CHS2(G(-3),G(-2),b,4,LEG) ;
  CH4(F(0),G(0),B(0),D(1),o,16/3,NON) ; CHS2(G(-3),G(-2),b,4,LEG) ;
   CH4(G(0),G(0),C(0),E(1),o,16,STAC) ;
    CH2(G(-2),G(-1),b,4,STAC) ; CHS4(F(0),G(0),B(0),D(1),o,2,LEG) ;
     CH2(G(-3),G(-2),b,4,NON) ;
CH4(E(0),G(0),C(1),C(1),o,4,NON) ; CHS2(C(-3),C(-2),b,4,NON) ;
CH4(C(1),E(1),G(1),C(2),o,4,STAC) ; CHS2(C(-2),G(-2),b,4,STAC) ;
   CHS3(C(1),E(0),G(0),t,3,STAC) ;
ter ;
```

Acknowledgment

The author thanks Gareth Loy and Lee Ray for careful and timely proofreading of this document.

The Phase Vocoder: A Tutorial

Mark Dolson

Computer Audio Research Laboratory
Center for Music Experiment, Q-037
University of California, San Diego
La Jolla, California 92093

ABSTRACT

The phase vocoder is a digital signal processing technique of potentially great musical significance. It can be used to perform very high fidelity time scaling, pitch transposition, and myriad other modifications of recorded sounds. In this tutorial, I attempt to explain the operation of the phase vocoder in terms that musicians can understand.

Introduction

For composers interested in the modification of natural sounds, the phase vocoder is a digital signal processing technique of potentially great significance. By itself, the phase vocoder can perform very high fidelity time-scale modification or pitch transposition of a wide range of sounds. In conjunction with a standard software synthesis program, the phase vocoder can provide the composer with arbitrary control of individual harmonics. But use of the phase vocoder to date has been limited primarily to experts in digital signal processing. Consequently, its musical potential has remained largely untapped.

In this article, I attempt to explain the operation of the phase vocoder in terms that musicians can understand. I rely heavily on the familiar concepts of sine waves, filters, and additive synthesis, and I employ a minimum of mathematics. My hope is that this tutorial will lay the groundwork for widespread use of the phase vocoder, both as a tool for sound analysis and modification, and as a catalyst for continued musical exploration.

Overview

Historically, the phase vocoder comes from a long line of voice coding techniques which were developed primarily for the electronic processing of speech. Indeed, the word "vocoder" is simply a contraction of the term "voice coder." There are many different types of vocoders. The *phase* vocoder was first described in 1966 in an article by Flanagan and Golden. However, it is only in the past ten years that this technique has really become popular and well understood.

The phase vocoder is one of a number of digital signal processing algorithms which can be categorized as analysis-synthesis techniques. Mathematically, these techniques are sophisticated algorithms which take an input signal and produce an output signal which is either identical to the input signal or a modified version of it. The underlying assumption is that the input signal can be well represented by a model whose parameters are varying with time. The analysis is devoted to determining the values of these parameters for the signal in question, and the synthesis is simply the output of the model itself. For example, in linear prediction the signal is modeled as the output of a time-varying filter whose input and frequency-response are determined by the analysis.

The benefits of analysis-synthesis formulations are considerable. Since the synthesis is based on an analysis of a specific signal, the synthesized output can be virtually identical to the original input; this can occur even when the signal in question bears little relation to the assumed model. Furthermore, the parameter values which are derived from the analysis can be modified to synthesize any number of useful modifications of the original signal. In this case, however, the perceptual significance and musical utility of the result depends critically on the degree to which the assumed model matches the signal to be modified.

In the phase vocoder, the signal is modeled as a sum of sine waves, and the parameters to be determined by analysis are the time-varying amplitudes and frequencies of each sine wave. Since these sine waves are not required to be harmonically related, this model is appropriate for a wide variety of musical signals. As a result, the phase vocoder can serve as the basis for a variety of very high fidelity modifications. It is this feature which makes the phase vocoder so useful in computer music.

In the sections which follow, I show in detail how the phase vocoder analysis-synthesis is actually performed. In particular, I show that there are two complementary (but mathematically equivalent) viewpoints which may be adopted. I refer to these as the *Filter Bank* interpretation and the *Fourier Transform* interpretation, and I discuss each in turn. Lastly, I show how the results of the phase vocoder analysis can be used musically to effect useful modifications of recorded sounds.

The Filter Bank Interpretation

The simplest view of the phase vocoder analysis is that it consists of a fixed bank of bandpass filters with the output of each filter expressed as a time-varying amplitude and a time-varying frequency (see Figure 1). The synthesis is then literally a sum of sine waves with the time-varying amplitude and frequency of each sine wave being obtained directly from the corresponding bandpass filter. If the center frequencies of the individual bandpass filters happen to align with the harmonics of a musical signal, then the outputs of the phase vocoder analysis are essentially the time-varying amplitudes and frequencies of each harmonic. But even when this situation does not obtain, the phase vocoder analysis is still surprisingly useful.

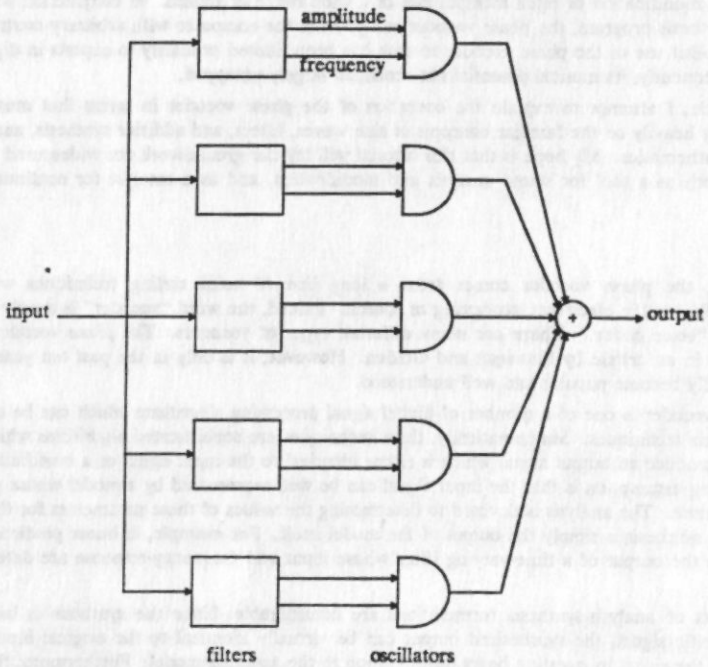


Figure 1. The Filter Bank Interpretation

The filter bank itself has only three constraints. First, the frequency response characteristics of the individual bandpass filters are identical except that each filter has its passband centered at a different frequency. Second, these center frequencies are equally spaced across the entire spectrum from 0 Hz to half the sampling rate. Third, the individual bandpass frequency response is such that the combined frequency response of all the filters in parallel is essentially flat across the entire spectrum. This ensures that no frequency component is given disproportionate weight in the analysis, and that the phase vocoder is in fact an analysis-synthesis identity. As a consequence of these constraints, the only issues in the design of the filter bank are the number of filters and the individual bandpass frequency response.

The number of filters must be sufficiently large so that there is never more than one partial within the passband of any single filter. For harmonic sounds, this amounts to saying that the number of filters must be greater than the sampling rate divided by the pitch. For inharmonic and polyphonic sounds, the number of filters may need to be much greater. If this condition is not satisfied, then the phase vocoder will not function as intended because the partials within a single filter will constructively and destructively interfere with each other, and the information about their individual frequencies will be coded as an unintended temporal variation in a single composite signal.

The design of the representative bandpass filter is dominated by a single consideration: the sharper the filter frequency response cuts off at the band edges (i.e., the less overlap between adjacent bandpass filters), the longer its impulse response will be (i.e., the longer the filter will "ring"). Thus, to get sharp cut-offs with minimal overlap, one must use filters whose time response is very sluggish. In the phase vocoder, this

tradeoff is ever-present, and the best solution is generally discovered experimentally by simply trying different filter settings for the sound in question.

A Closer Look at the Filter Bank

The above paragraphs provide an adequate description of the phase vocoder from the standpoint of the user, but they leave unanswered the question of how it actually works. In this section, I show in detail how the output of a single bandpass filter is expressed as a time-varying amplitude and a time-varying frequency.

The actual operation of a single phase-vocoder bandpass filter is shown in Figure 2. This diagram may appear complicated, but it can easily be broken down into a series of fairly simple mathematical steps.

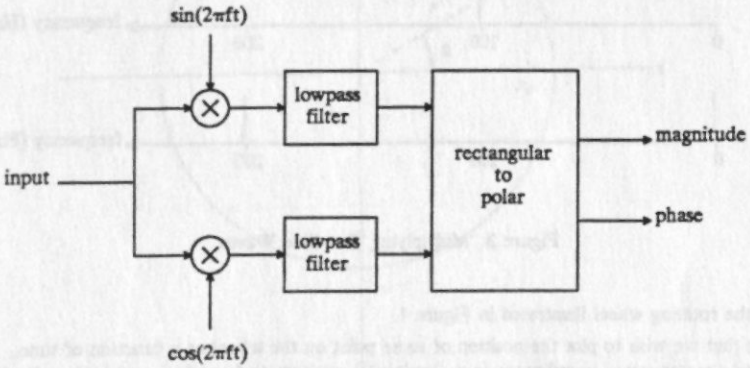


Figure 2. An Individual Bandpass Filter

In the first step, the incoming signal is routed into two parallel paths. In one path, the signal is multiplied by a sine wave with an amplitude of 1.0 and a frequency equal to the center frequency of the bandpass filter; in the other path, the signal is multiplied by a cosine wave of the same amplitude and frequency. Thus, the two parallel paths are identical except for the phase of the multiplying waveform. Then, in each path, the result of the multiplication is fed into a lowpass filter.

The multiplication operation itself should be familiar to musicians as simple ring modulation. Multiplying any signal by a sine (or cosine) wave of constant frequency has the effect of simultaneously shifting all the frequency components in the original signal by both plus and minus the frequency of the sine wave. An example of this is shown in Figure 3 in which a 100 Hz sine wave multiplies an input signal of 101 Hz. The result is a sine wave at 1 Hz (i.e., 101 Hz – 100 Hz) and a sine wave at 201 Hz (i.e., 101 Hz + 100 Hz). Furthermore, if this result is now passed through an appropriate lowpass filter, only the 1 Hz sine wave will remain. This sequence of operations (i.e., multiplying by a sine wave of frequency f and then lowpass filtering) is useful in a variety of signal processing applications and is known as *heterodyning*. Any input frequency components in the vicinity of frequency f are shifted down to the vicinity of 0 Hz and allowed to pass; input frequency components not in the vicinity of frequency f are similarly shifted but not by enough to get through the lowpass filter. The result is a type of bandpass filtering in which the passband is frequency-shifted down to very low frequencies.

In the phase vocoder, heterodyning is performed in each of the two parallel paths. But since one path heterodynes with a sine wave while the other path uses a cosine wave, the resulting heterodyned signals in the two paths are out of phase by 90 degrees. Thus, in the above example, both paths will produce a 1 Hz sinusoidal wave at the outputs of their respective lowpass filters, but the two sinusoids will be 90 degrees out of phase with respect to each other. To understand what the phase vocoder does next with these signals, we

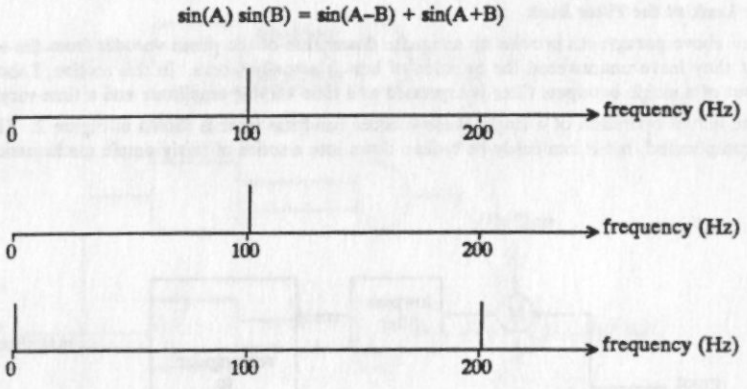


Figure 3. Multiplying Two Sine Waves

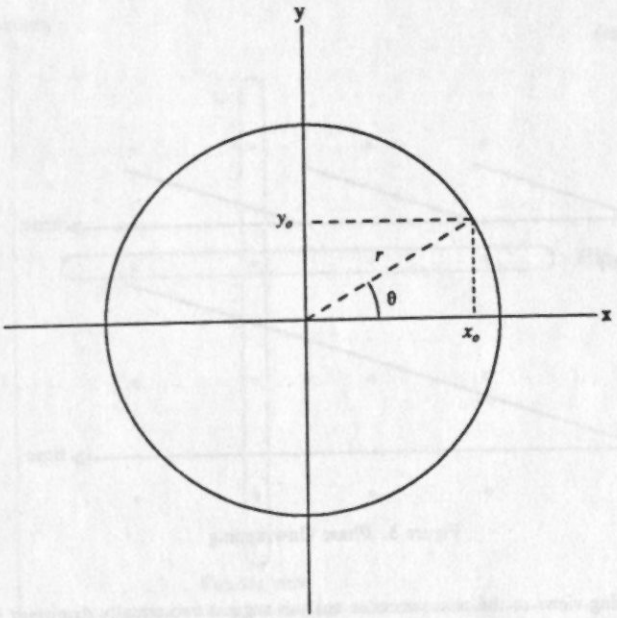
can consider the rotating wheel illustrated in Figure 4.

Suppose that we wish to plot the position of some point on the wheel as a function of time. We have a choice of using "rectangular" coordinates (e.g., horizontal position and vertical position) or "polar" coordinates (e.g., radial position and angular position—also known as magnitude and phase). With rectangular coordinates we find that both the horizontal position and the vertical position are varying sinusoidally, but the maximum vertical displacement occurs one quarter cycle later than the maximum horizontal displacement. With polar coordinates we simply have a linearly increasing angular position and a constant radius. Clearly, the latter description is simpler.

The situation within the phase vocoder is very much analogous. The two heterodyned signals can be viewed as the horizontal and vertical signals of the rectangular representation, whereas the desired representation is in terms of a time-varying amplitude (i.e., radius) and a time-varying frequency (i.e., rate of angular rotation). Happily, the translation between the two different representations is easily accomplished. As shown in Figure 4, the amplitude at each point in time is simply the square root of the sum of the squares of the two rectangular coordinates. The frequency cannot be calculated directly, but it can be very well approximated by taking the difference in successive values of angular position and then dividing by the time between these successive values. To see this, we can note that the difference between two successive values of angular position is some fraction of an entire cycle (i.e., a complete revolution), and that "frequency" is simply the number of cycles which occur during some unit time interval. As a result, we need only worry about how to calculate the angular position.

Figure 4 also gives a formula for the angular position, but it produces answers only in the range of 0 to 360 degrees. Thus, if we examine successive values of angular position, we may find a sequence such as 180, 225, 270, 315, 0, 45, 90. This suggests that the instantaneous frequency (i.e., rate of angular rotation) is given by the sequence: $(225 - 180)/T = 45/T$, $(270 - 225)/T = 45/T$, $(315 - 270)/T = 45/T$, $(0 - 315)/T = -315/T$, $(45 - 0)/T = 45/T$, $(90 - 45)/T = 45/T$, where T is time between successive values. But the $-315/T$ element is clearly not quite right.

What has actually happened is that we have gone through more than a single cycle. Therefore, if we want our frequency calculation to work properly, we should really write the sequence as 180, 225, 270, 315, 360, 405, 450. Now the result of the frequency calculation is, as it should be, a sequence of (45/T)'s. This process of adding in 360 degrees whenever a full cycle has been completed is known as *phase unwrapping* (see Figure 5). It is the final necessary step in the sequence of operations which makes the phase vocoder work.



$$r = \sqrt{x_o^2 + y_o^2}$$

$$\theta = arctan\left(\frac{y_o}{x_o}\right)$$

Figure 4. Rectangular and Polar Coordinates

Thus, the internal operation of a single phase vocoder bandpass filter consists of (1) heterodyning the input with both a sine wave and a cosine wave in parallel, (2) lowpass filtering each result, (3) converting the two parallel lowpass filtered signals to radius and angular-position signals, (4) unwrapping the angular-position values, and (5) subtracting successive unwrapped angular-position values and dividing by the time to obtain a rate-of-angular-rotation signal. But it should be noted that this rate-of-rotation signal (i.e., the instantaneous frequency) actually refers only to the difference frequency between the heterodyning sinusoid (i.e., the filter center frequency) and the input signal. Therefore the final step is simply to add the filter center frequency back in.

The Fourier Transform Interpretation

A complementary (and equally correct) view of the phase-vocoder analysis is that it consists of a succession of overlapping Fourier transforms taken over finite-duration windows in time. It is interesting to compare this perspective to that of the Filter Bank interpretation. In the latter, the emphasis is on the temporal succession of magnitude and phase values in a single filter band. In contrast, the Fourier Transform interpretation focuses attention on the magnitude and phase values for all of the different filter bands or *frequency bins* at a single point in time (see Figure 6).

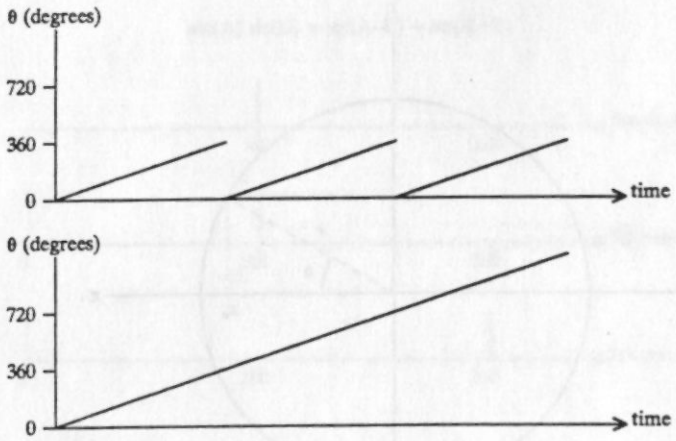


Figure 5. Phase Unwrapping

These two differing views of the phase-vocoder analysis suggest two equally divergent interpretations of the resynthesis. In the Filter Bank interpretation (as noted above), the resynthesis can be viewed as a classic example of additive synthesis with time-varying amplitude and frequency controls for each oscillator. In the Fourier view, the synthesis is accomplished by converting back to real-and-imaginary form and overlap-adding the successive inverse Fourier transforms. This is a first indication that the phase vocoder representation may actually be more generally applicable than would be expected of an additive-synthesis technique.

In the Fourier interpretation, the number of filters bands in the phase vocoder is simply the number of points in the Fourier transform. Similarly, the equal spacing in frequency of the individual filters can be recognized as a fundamental feature of the Fourier transform. On the other hand, the shape of the filter passbands (e.g., the steepness of the cutoff at the band edges) is determined by the shape of the window function which is applied prior to calculating the transform. For a particular characteristic shape (e.g., a Hamming window), the steepness of the filter cutoff increases in direct proportion to the duration of the window. Thus, again, we see the fundamental tradeoff between rapid time response and narrow frequency response.

It is important to understand that the two different interpretations of the phase vocoder analysis apply only to the implementation of the bank of bandpass filters. The operation (described in the previous section) by which the outputs of these filters are expressed as time-varying amplitudes and frequencies is the same for each. However, a particular advantage of the Fourier interpretation is that it leads to the implementation of the filter bank via the more efficient Fast Fourier Transform (FFT) technique. The FFT produces an output value for each of N filters with (on the order of) $N \log_2 N$ multiplications, while the direct implementation of the filter bank requires N^2 multiplications. Thus, the Fourier interpretation can lead to a substantial increase in computational efficiency when the number of filters is large (e.g., $N = 1024$).

The Fourier interpretation has also been the key to much of the recent progress in phase-vocoder-like techniques. Mathematically, these techniques are described as Short-Time Fourier-Transform techniques [Rabiner & Schafer, 1978; Crochiere, 1980; Portnoff, 1980; Portnoff, 1981a,b; Griffin & Lim, 1984]. Such algorithms may also be referred to as Multirate Digital Signal Processing techniques (for reasons which will be made clear below) [Crochiere & Rabiner, 1983].

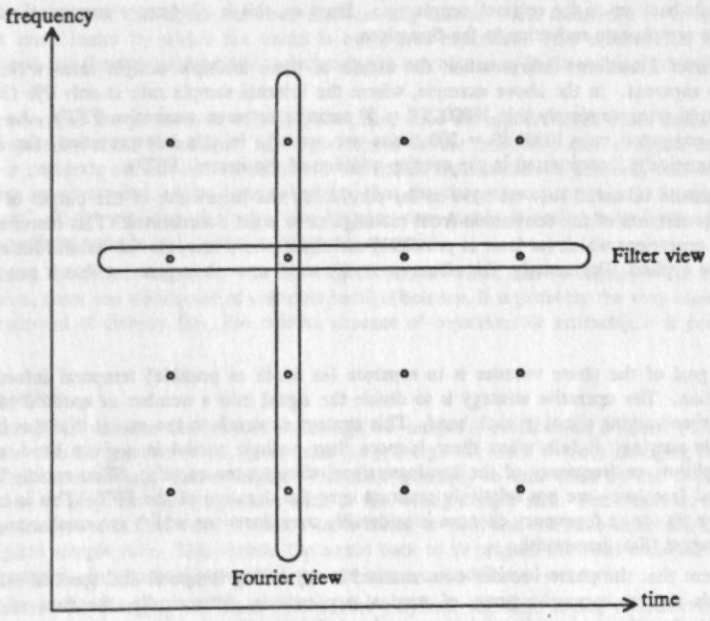


Figure 6. Filter Bank Interpretation vs. Fourier Transform Interpretation

Sample-Rate Considerations

The input and output signals to and from the phase vocoder are always assumed to be digital signals with a sampling rate of at least twice the highest frequency in the associated analog signal (e.g., a speech signal with a highest frequency of 5 KHz might be digitized—at least in principle—at 10 KHz and fed into the phase vocoder). However, the sample rates within the individual filter bands of the phase vocoder do not need to be nearly so high. This is most easily understood via the Filter Bank interpretation.

Within any given filter band, the result of the heterodyning and lowpass filtering operation is a signal whose highest frequency is equal to the cutoff frequency of the lowpass filter. For instance in the above example, the lowpass filter may only pass frequencies up to 50 Hz. Thus, although the input to the filter was a speech signal sampled at 10 KHz, the output of the filter can be sampled (at least in the ideal case) at as little as 100 Hz without any aliasing error. This is true for each of the bandpass filters, because each filter operates by heterodyning a certain frequency region down to the 0 – 50 Hz region.

In practice, the lowpass filter can never have an infinitely steep cutoff. Therefore to really avoid aliasing error, it is advisable to sample the output of the filter at four times the cutoff frequency (e.g., 200 Hz) as opposed to two. Still, this represents an enormous savings in computation (e.g., the filter output is calculated 200 times per second instead of 10,000 times). A detail worth noting here is that this savings is only possible because the filter is a finite impulse response (FIR) filter, (i.e., the present output is calculated entirely on the basis of present and past inputs).

If we now seek to resynthesize the original input from the phase vocoder analysis signals, we face a minor problem. The analysis signals (which in the Filter Bank interpretation are thought of as providing the instantaneous amplitude and frequency values for a bank of sine-wave oscillators) are no longer at the same

sample rate as the desired output signal. Thus, an additional interpolation operation is required to convert the analysis signals back up to the original sample rate. Even so, this is a lot more computationally efficient than avoiding the sample-rate reduction in the first place.

In the Fourier Transform interpretation the details of these multiple sample rates within the phase vocoder are less apparent. In the above example, where the internal sample rate is only 2% (200/10000) of the external sample rate, we simply skip 10000/200 = 50 samples between successive FFT's. As a result, the FFT values are computed only 10000/50 = 200 times per second. In this interpretation, the interpolation operation is automatically incorporated in the overlap-addition of the inverse FFT's.

Lastly, it should be noted that we have so far considered the bandwidth of the output of the lowpass filter without any mention of the conversion from rectangular to polar coordinates. This conversion involves highly nonlinear operations which (at least in principle) can significantly increase the bandwidth of the signals to which they are applied. Fortunately, this effect is usually small enough in practice that it can generally be ignored.

Applications

The basic goal of the phase vocoder is to separate (as much as possible) temporal information from spectral information. The operative strategy is to divide the signal into a number of spectral bands, and to characterize the time-varying signal in each band. This strategy succeeds to the extent that this bandpass signal is itself slowly varying. It fails when there is more than a single partial in a given band, or when the time-varying amplitude or frequency of the bandpass signal changes too rapidly. "Too rapidly" means that the amplitude and frequency are not relatively constant over the duration of the FFT. This is equivalent to saying that the amplitude or frequency changes considerably over durations which are small compared to the inverse of the lowpass filter bandwidth.

To the extent that the phase vocoder does succeed in separating temporal and spectral information, it provides the basis for an impressive array of musical applications. Historically, the first of these to be explored was that of analyzing instrumental tones to determine the time-varying amplitudes and frequencies of individual partials. This application was pioneered by Moorer and Grey at Stanford in the mid '70's in a landmark series of investigations of the perception of timbre [Grey & Moorer, 1977; Grey, 1977; Grey & Gordon, 1978; Moorer, 1978]. (The "heterodyne filter" technique developed by Moorer is essentially a special case of the phase vocoder.)

More recently, interest in the phase vocoder has focused more on its ability to modify and transform recorded sound materials in musically useful ways. The possibilities in this realm are myriad. However, two basic operations stand out as particularly significant. These operations are *time scaling* and *pitch transposition*.

Time Scaling

It is always possible to slow down a recorded sound simply by playing it back at a lower sample rate; this is analogous to playing a tape recording at a lower playback speed. But this kind of simplistic time expansion simultaneously lowers the pitch by the same factor as the time expansion. Slowing down the temporal evolution of a sound without altering its pitch requires an explicit separation of temporal and spectral information. As noted above, this is precisely what the phase vocoder attempts to do.

To understand the use of the phase vocoder for time scaling, it is helpful to once again consider the two basic interpretations described above. In the Filter Bank interpretation, the operation is simplicity itself. The time-varying amplitude and frequency signals for each oscillator are control signals which (hopefully) carry only temporal information. Stretching out these control signals (via interpolation) does not change the frequency of the individual oscillators at all, but it does slow down the temporal evolution of the composite sound. The result is a time-expanded sound with the original pitch.

The Fourier transform view of time scaling is more complicated, but it is no less instructive. The basic idea is that in order to time-expand a sound, the inverse FFT's can simply be spaced further apart than the analysis FFT's. As a result, spectral changes occur more slowly in the synthesized sound than in the original. But this overlooks the details of the magnitude and phase signals in the middle.

Consider a single bin within the FFT for which successive phase values are incremented by 45 degrees. This implies that the signal within that filter band is increasing in phase at a rate of 1/8 cycle (i.e., 45

degrees) per time interval, where the time interval in question is the time between successive FFT's. Spacing the inverse FFT's further apart means that the 45 degree increase now occurs over a longer time interval. Hence, the frequency of the signal has been inadvertently altered. The solution is to rescale the phase by precisely the same factor by which the sound is being time-expanded. This ensures that the signal in any given filter band has the same frequency variation in the resynthesis as in the original (though it occurs more slowly).

The reason that the problem of rescaling the phase does not appear in the Filter Bank interpretation is that the interpolation there is assumed to be performed on the frequency control signal as opposed to the phase. This is perfectly correct conceptually, but the actual implementation generally conforms more closely to the Fourier interpretation. Also, by emphasizing that the time expansion amounts to spacing out successive "snapshots" of the evolving spectrum, the Fourier view makes it easier to understand how the phase vocoder can perform equally well with non-harmonic material.

Of course, the phase vocoder is not the only technique which can be employed for this kind of time scaling. Indeed, from the standpoint of computational efficiency, it is probably the very least attractive. But from the standpoint of fidelity (i.e., the relative absence of objectionable artifacts), it is decidedly the most desirable.

Pitch Transposition

Since the phase vocoder can be used to change the temporal evolution of a sound without changing its pitch, it should also be possible to do the reverse (i.e., change the pitch without changing the duration). In fact, this operation is trivially accomplished. The trick is simply to time scale by the desired pitch-change factor, and then to play the resulting sound back at the wrong sample rate. For example, to raise the pitch by an octave, the sound is first time-expanded by a factor of two, and the time-expansion is then played at twice the original sample rate. This shrinks the sound back to its original duration while simultaneously doubling all frequencies. In practice, however, there are also some additional concerns.

First, instead of changing the clock rate on the playback digital-to-analog converters, it is more convenient to simply do a sample-rate conversion on the time-scaled sound via software. Thus, in the above example, we would simply designate a higher sample rate for the time-expanded sound, and then sample-rate convert it down by a factor of two so that it could be played at the normal sample rate. It is possible to embed this sample-rate conversion within the phase vocoder itself, but this proves to be of only marginal utility and will not be further discussed.

Second, upon closer examination it can be seen that only time-scale factors which are ratios of integers are actually allowed. This is clearest in the Fourier view because the expansion factor is simply the ratio of the number of samples between successive analysis FFT's to the number of samples between successive synthesis FFT's. However, it is equally true of the Filter Bank interpretation because it turns out that the control signals can only be interpolated by factors which are ratios of two integers. Of course, this has little significance for time scaling because, while it may be impossible to find two suitable integers with precisely the desired ratio, the error is perceptually negligible. However, when time scaling is performed as a prelude to pitch transposition, the perceptual consequences of such errors are greatly magnified (by virtue of the ear's sensitivity to small pitch differences), and considerable care may be required in the selection of two appropriate integers.

An additional complication arises when modifying the pitch of speech signals because the transposition process changes not only the pitch, but also the frequency of the vocal tract resonances (i.e., the formants). For shifts of an octave or more, this considerably reduces the intelligibility of the speech. (This same phenomena occurs in the pitch transposition of non-speech sounds as well, but for these sounds intelligibility is not an issue.) To correct for this, an additional operation may be inserted into the phase vocoder algorithm as shown in Figure 7. For each FFT, this additional operation determines the spectral envelope (i.e., the shape traced out by the peaks of the harmonics as a function of frequency), and then distorts this envelope in such a way that the subsequent sample-rate conversion brings it back precisely to its original shape.

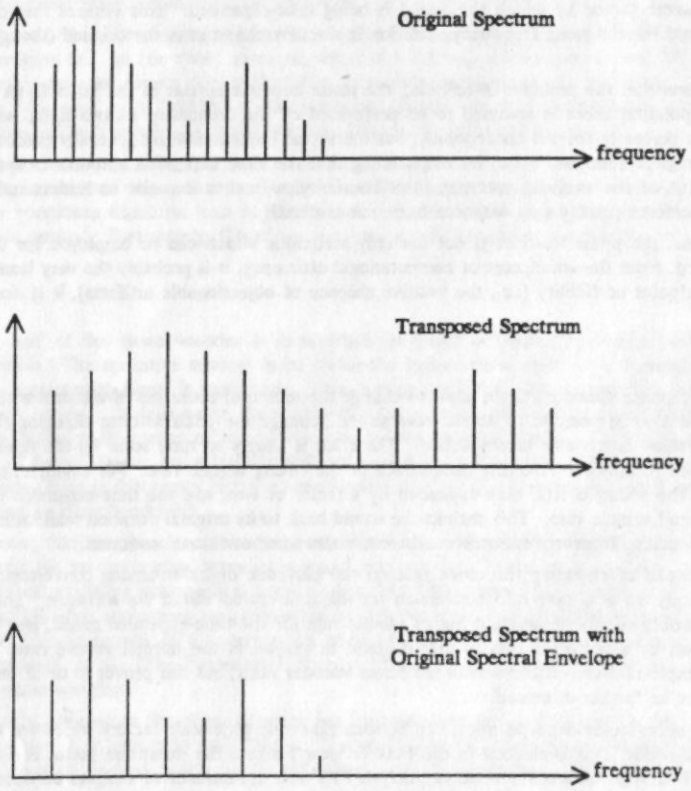


Figure 7. Spectral Envelope Correction

Summary

The above descriptions address only the most elementary possibilities of the phase vocoder technique. In addition to simple time scaling and pitch transposition, it is also possible to perform time-varying time scaling and pitch transposition, time-varying filtering (e.g., cross synthesis), and nonlinear filtering (e.g., noise reduction), all with very high fidelity. A detailed description of these applications, and of their musical implications, is the subject of a separate paper.

References

Crochiere, R. E. (1980). A weighted overlap-add method of Fourier analysis-synthesis. *IEEE Transactions on Acoustics, Speech, and Signal Processing, ASSP-28*(1), 55-69.

Crochiere, R. E. & Rabiner, L. R. (1983). *Multirate Digital Signal Processing*. Englewood Cliffs, NJ: Prentice-Hall.

Flanagan, J. L. & Golden, R. M. Phase vocoder. *Bell System Technical Journal*, *45*, 1493-1509.

Grey, J. M., & Moorer, J. A. (1977). Perceptual evaluations of synthesized musical instrument tones. *Journal of the Acoustical Society of America*, *62*, 454-462.

Grey, J. M. (1977). Multidimensional perceptual scaling of musical timbres. *Journal of the Acoustical Society of America*, *61*, 1270-1277.

Grey, J. M., & Gordon, J. W. (1978). Perceptual effects of spectral modifications on musical timbres. *Journal of the Acoustical Society of America*, *63*, 1493-1500.

Griffin, D. W. & Lim, J. S. (1984). Signal estimation from modified short-time Fourier transform. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *ASSP-28*(2), 236-242.

Moorer, J. A. (1978) The use of the phase vocoder in computer music applications. *Journal of the Audio Engineering Society*, *24*(9), 717-727.

Portnoff, M. R. (1980). Time-frequency representation of digital signals and systems based on short-time Fourier analysis. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *ASSP-28*(1), 55-69.

Portnoff, M. R. (1981a). Short-time Fourier analysis of sampled speech. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *ASSP-29*(3), 364-373.

Portnoff, M. R. (1981b). Time-scale modification of speech based on short-time Fourier analysis. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, *ASSP-29*(3) 374-390.

A Selected, Annotated Bibliography for Computer Music

F. Richard Moore

Computer Audio Research Laboratory
Center for Music Experiment, Q-037
University of California, San Diego
La Jolla, California 92093

ABSTRACT

This bibliography cites and briefly describes references taken from the literature of computer music and related fields. It is by no means complete, and is intended to act only as a guide for the newcomer to computer music. Many of the references can be found in two recently published books: *Digital Audio Signal Processing: An Anthology*, John Strawn, ed., William Kaufmann, Inc. (1985), and *Foundations of Computer Music*, Curtis Roads and John Strawn, eds., MIT Press (1985). These anthologies bring together many articles that have been published over the past decade or so in a book form of unprecedented convenience in the field of computer music. Most of the articles in the anthologies come from *Computer Music Journal*—still the best single source for information about computer music. Annotations are provided for certain of the references—in other cases merely the title should suffice.

1. Background

The following references provide general background information in computer music, mathematics, digital signal processing, computer programming, and psychoacoustics.

1.1. General

D. G. Loy, "The Composer Seduced into Programming," *Perspectives of New Music* **19**(1-2) pp. 184-198 (Fall-Winter 1980, Spring-Summer 1981).

A general account of the likely effect of computers on the nature of musicians.

M. V. Mathews, F. R. Moore, and J.-C. Risset, "Computers and Future Music," *Science* **183** pp. 263-268 (25 Jan 1974).

A general account of the likely effect of computers on the nature of music.

M. V. Mathews with the collaboration of J. E. Miller, F. R. Moore, J.-C. Risset, and J. R. Pierce, *The Technology of Computer Music*, MIT Press (1969).

This book includes the original published description of the MUSIC V program: a general introduction to digital audio, a tutorial section with many examples of MUSIC V use, and the MUSIC V manual.

F. R. Moore, "The Futures of Music," *Perspectives of New Music* **19**(1-2) pp. 212-226 (Fall-Winter 1980, Spring-Summer 1981).

This article discusses the effect on musical possibilities of rapidly changing integrated circuit technology.

F. R. Moore, "The Computer Audio Research Laboratory at UCSD," *Computer Music Journal* 6(1) pp. 18-29 (Spring 1982).

This articles describes the overall design of the CARL system as it was originally conceived.

F. R. Moore, "The Computer Music Journal, 1977–1982—a Review," *Yale Journal of Music Theory* **27.1** pp. 127-135 (Spring 1983).

A critical review of the early days of the Computer Music Journal.

1.2. Mathematics

R. C. Fisher and A. D. Ziebur, *Integrated Algebra, Trigonometry, and Analytic Geometry*, Prentice-Hall, Inc. (1982).

A good general math review book written by expert textbook authors.

F. R. Moore, "An Introduction to the Mathematics of Digital Signal Processing, Part I: Algebra, Trigonometry, and the Most Beautiful Formula in Mathematics," in *Digital Audio Signal Processing: An Anthology*, ed. J. Strawn, William Kaufmann, Inc. (Computer Music Book Series) (1985).

The simplest possible set of background math for digital audio signal processing written especially for musical readers.

M. R. Spiegel, *Mathematical Handbook*, McGraw-Hill Book Company (1968).

This is not a textbook but a general handbook of formulas and relations taken from many subfields of mathematics.

1.3. Psychoacoustics

The following references provide background material on psychoacoutics relevant to music.

J. R. Pierce, *The Science of Musical Sound*, Scientific American Books, Inc. (distributed by W. H. Freeman and Company) (1983).

Not a textbook but written in the style of a gala Scientific American *article, this book provides an authoritative and comprehensive survey of recent findings in psychoacoustics, many of which are relevant to music.*

J. G. Roederer, *Introduction to the Physics and Psychophysics of Music*, Springer-Verlag (1975).

A good general reference for musical psychoacoustics.

1.4. Programming

B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc. (1978).

The "bible" for C programmers, this book is intended for readers who already know how to program and wish to learn the C language.

H. McGilton and R. Morgan, *Introducing the UNIX System*, McGraw-Hill (1983).

This book describes the Berkeley-enhanced UNIX operating system (the one used at CARL) including the file system, commands, inter-user communication, text manipulation (including ed, ex, vi, awk, and sed), and document formatting.

B. W. Kernighan and R. Pike, *The UNIX Programming Environment*, Prentice-Hall, Inc. (1984).

This book describes the Bell Laboratories version of UNIX, without Berkeley enhancements such as vi and the cshell.

D. A. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley (1968).

An indispensible guide to basic programming methodology regardless of the particular language in which these ideas are expressed.

F. R. Moore, *Programming in C with a Bit of UNIX*, Prentice-Hall, Inc. (1985).

An introduction to programming for the beginner who wishes to become computer literate without becoming a computer scientist.

1.5. Digital Signal Processing

F. R. Moore, "An Introduction to the Mathematics of Digital Signal Processing, Part II: Sampling, Transforms, and Digital Filtering," in *Digital Audio Signal Processing: An Anthology*, ed. J. Strawn, William Kaufmann, Inc. (Computer Music Book Series) (1985).

A basic introduction to the ideas of digital audio processing written especially for musicians.

J. A. Moorer, "Signal Processing Aspects of Computer Music: A Survey," in *Digital Audio Signal Processing: An Anthology*, ed. J. Strawn, William Kaufmann, Inc. (Computer Music Book Series) (1985).

An excellent survey of the basic signal processing techniques used in computer music as of its original publication date (1977).

L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, Inc. (1978).

A comprehensive treatment of the application of digital signal processing techniques to the analysis and synthesis of speech.

L. R. Rabiner and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall, Inc. (1975).

The "bible" of digital signal processing for many engineers, this excellent textbook provides a thorough treatment of all fundamentals of digital signal processing independent of their application.

J. O. Smith, "An Introduction to Digital Filter Theory," in *Digital Audio Signal Processing: An Anthology*, ed. J. Strawn, William Kaufmann, Inc. (Computer Music Book Series) (1985).

A basic introduction to the ideas of digital filtering written expressly for musicians.

2. Digital Sound Synthesis Techniques

The following references provide basic information on sound synthesis techniques used in computer music.

2.1. General

R. Cann, "An Analysis/Synthesis Tutorial," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

2.2. Linear Synthesis Techniques

J. W. Gordon and J. Strawn, "An Introduction to the Phase Vocoder," in *Digital Audio Signal Processing: An Anthology*, ed. J. Strawn, William Kaufmann, Inc. (Computer Music Book Series) (1985).

A good general explanation of the operation of the phase vocoder.

L. Hiller and P. Ruiz, "Synthesizing Musical Sounds by Solving the Wave Equation for Vibrating Objects, I & II," *Journal of the Audio Engineering Society* 19 pp. 462-470, 542-551 (1971).

One of the few early attempts to synthesize sounds via physical modeling techniques.

C. Roads, "Granular Synthesis of Sound," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

Granular synthesis involves the concatenation and combination of very short sonic "granules" in an as-yet-to-be-thoroughly-explored way.

T. L. Petersen, "Spiral Synthesis," in *Digital Audio Signal Processing: An Anthology*, ed. J. Strawn, William Kaufmann, Inc. (Computer Music Book Series) (1985).

A novel synthesis method based on projections of spirals in the complex number domain onto real axes.

2.3. Nonlinear Synthesis Techniques

J. Beauchamp, "Brass-Tone Synthesis by Spectrum Evolution Matching with Nonlinear Functions," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

A novel use of nonlinear mapping functions to achieve control over the brightness of a timbre applied to brass tone synthesis.

D. Morrill, "Trumpet Algorithms for Computer Composition," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

A description of the FM-based methods discovered by Morrill to synthesize realistic trumpet-like sonorities.

C. Roads, "A Tutorial on Nonlinear Distortion or Waveshaping Synthesis," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

A tutorial paper describing the use of sums of Chebychev polynomial functions to provide nonlinear mappings between sinusoids and arbitrary target spectra.

2.3.1. Frequency Modulation

J. M. Chowning, "The Synthesis of Complex Audio Spectra by Means of Frequency Modulation," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

The classic paper that introduced the basic ideas of frequency modulation as a surprisingly rich method for synthesizing musical sounds.

M. LeBrun, "A Derivation of the Spectrum of FM with a Complex Modulating Wave," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

B. Schottstaedt, "The Simulation of Natural Instrument Tones Using Frequency Modulation with a Complex Modulating Wave," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

B. Truax, "Organizational Techniques for c:m Ratios in Frequency Modulation," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

2.4. Spatial Sound Processing

J. M. Chowning, "The Simulation of Moving Sound Sources," *Journal of the Audio Engineering Society* **19**(1)(Jan 1971).

Another classic paper on the spatialization of sounds in illusory acoustic space.

F. R. Moore, "A General Model for Spatial Processing of Sounds," *Computer Music Journal* 7(3) pp. 6-15 (Fall 1983).

A full explication of the physical models and concepts upon which the cmusic space unit generator is based.

J. A. Moorer, "About This Reverberation Business," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

A good general treatment of digital reverberation and its relation to the physics of concert halls and perception.

3. Synthesizers

The following references provide background material on techniques used in digital music synthesizers.

H. Alles, "A Portable Digital Sound-Synthesis System," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

The introduction to a series of papers on the world's first all-digital, standalone, programmable music synthesis system.

H. Alles and G. diGiugno, "The 4B: A One-Card 64-Channel Digital Synthesizer," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

Description of a precursor to the synthesis engine module that forms the heart of the IRCAM 4X machine.

H. Alles, "A 256-Channel Performance-Input Device," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

F. R. Moore, "Table Lookup Noise for Sinusoidal Digital Oscillators," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

This paper explores the distortion tradeoffs in table length and width for table lookup digital oscillators of the truncating, rounding, and interpolating types.

F. R. Moore, "The FRMbox: A Modular Digital Music Synthesizer," in *Computer Music Book Series*, ed. J. Strawn, William Kaufmann, Inc. (forthcoming in 1985).

J. Snell, "Design of a Digital Oscillator That Will Generate up to 256 Low-Distortion Sine Waves in Real Time," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

4. Software Methodology

The following references provide background material on software techniques used in computer music.

C. Abbott, "Automated Microprogramming for Digital Synthesizers," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

C. Abbott, "A Software Approach to Interactive Processing of Musical Sound," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

J. Chadabe and R. Meyers, "An Introduction to the PLAY Program," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

Description of the author's software for realtime interactive computer-aided composition.

M. V. Mathews and F. R. Moore, "GROOVE—A Program to Compose, Store, and Edit Functions of Time," *Communications of the ACM* 13(12) pp. 715-721 (Dec. 1970).

The only published account of the GROOVE system which provided general realtime control of analog sound synthesis equipment.

F. R. Moore, "Musical Signal Processing in a UNIX Environment," pp. 89-111 in *Proceedings of the International Music and Technology Conference*, University of Melbourne, Australia (Aug. 1981).

A basic description of the ways in which UNIX pipe mechanisms may be used to interlink signal processing programs.

F. R. Moore, "Computer-Controlled Analog Synthesizers," Bell Laboratories Computing Science Technical Report (May 1973).

A general description of the problems involved in computer control of music synthesizers.

J. Myhill, "Controlled Indeterminacy: A First Step Toward a Semistochastic Music Language," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

C. Roads, "Grammars as Representations for Music," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

5. Musical Acoustics and Psychoacoustics

The following references provide background material on psychoacoustics as applied to computer music.

S. McAdams and A. Bregman, "Hearing Musical Streams," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

J.-C. Risset and D. Wessel, "Exploration of Timbre by Analysis and Synthesis," pp. 26-58 in *The Psychology of Music*, ed. D. Deutsch, Academic Press (1982).

D. Wessel, "Timbre Space as a Musical Control Structure," in *Foundations of Computer Music*, ed. C. Roads and J. Strawn, MIT Press (1985).

Musical Signal Processing in a UNIX Environment

- F. Richard Moore, CARL Director

Background

UNIX is well known as a good operating system for wordprocessing and program development[2, 3, 13]. Its properties for digital signal processing applications are less well understood. This paper presents techniques for applying UNIX to digital signal processing at the operating system level, with applications to musical signals developed at UCSD's Computer Audio Research Laboratory (CARL). It is necessarily assumed that the reader is familiar with UNIX[12], C programming[4, 11], and digital signal processing techniques[5, 8, 10], since the purpose of this paper is to relate these three.

Signal Processing on Computers

Digital signal processing by computer has many inherent advantages, not the least of which is the flexibility with which programs may be written, changed, and disseminated. Signal processing algorithms are invariably studied and developed as computer programs before they are implemented in any other form, if they ever are. Signal processing also finds a wide variety of applications in a broad class of fields, including music and speech, as well as radar, radiotelescopy, seismology and medicine.

Probably the most salient characteristic of a digital signal from a computer processing standpoint is its bandwidth. For seismological work, sampling rates are usually low enough to allow realtime computer processing of data. For most other work, including music and speech, the bandwidth is high enough to disallow realtime processing, even for the simplest algorithms. This means that the efficiency of the processing method is an important concern. The major historical breakthrough in digital signal processing was the discovery of the so-called fast Fourier transform (FFT) by J. W. Cooley and J. W. Tukey in 1965[1]. This algorithm transformed digital signal processing from a theoretical possibility to a practical reality by increasing the computational efficiency of Fourier transformation by several orders of magnitude in many cases.

Typical audio signal bandwidth for audio work at CARL is about 50,000 samples per second, for each audio channel, each sample expressed to a precision of 16 bits. For signal processing, it is common to use 32-bit floating point representation for the sample values, especially when IIR filtering is used, since the dynamic range of intermediate values in such computations can vary widely.

Thus the audio signals for a quadraphonic audio signal at CARL have a typical worst case bandwidth of (50,000 samples/second) times (32 bits per sample) times (4 channels) = (6.4 million bits per second of sound). Obviously, even the most efficient algorithms running on modern computers will run quite slowly when so much data must be processed. Array processors are used to advantage for standard signal modification tasks, such as filtering or FFTs, but music synthesis algorithms are often not within their capabilities, rendering them of limited value for music work. The only realistic solution to the problem of achieving realtime music capability remains the construction

of special-purpose hardware, optimized around the signal processing needs of music[6, 7].

UNIX and the C Programming Language

The C programming language is a major feature of UNIX. It can be characterized as a general-purpose, fairly terse language, with modern control flow and data structures, a rich set of operators, and a large and growing following. The UNIX operating system itself is written almost exclusively in C, which is a measure of the fine level of control possible with a such a "not-so-high-level" language.

From the signal processing standpoint, the C language is certainly preferable to work with compared to the most common signal processing language, FORTRAN. The control and data structures of C alone would justify its use for many projects. However, FORTRAN is so well-entrenched in the signal processing community that it is not likely to disappear soon as a popular choice for new applications.

The biggest advantages of FORTRAN are its ubiquity and its efficiency. FORTRAN is very well known, and it is probably the only reliably available language which exists for signal processing. FORTRAN is also well optimized around the numerical tasks which typify signal processing calculations. FORTRAN is, however, relatively difficult to work with, since the programming and debugging time it requires are often greater than with languages featuring more modern control structures. Some dialects of FORTRAN, notably FORTRAN 77, are attempts to superimpose reasonable control structures on FORTRAN.

The C language is certainly no harder to use than FORTRAN for signal processing programs. Of course, C is not perfect either. One unfortunate characteristic of C is that, like many "high-level" languages, it uses double precision arithmetic for all internal calculations (this is a specification of the language). Such precision is obtained at a cost in running time, sometimes a severe one, making it desirable for the C compiler to allow optional use of single precision where the user deems this admissible. Another minor inconvenience in C that a FORTRAN programmer would surely notice is the lack of an exponentiation operator (the FORTRAN statement "X = Y**Z" is translated into C as "x = pow(y, z);"). This C-ism is no less efficient than FORTRAN, but exponents are common in numerical work, and it represents a noticeable inconvenience to have to explicitly call a subprogram for them in some programs. Aside from these drawbacks, however, the degree of control that C makes possible over the machine operations allows computational strategies to be fine-tuned. It is precisely this fine-tuning capability which makes it possible to write a complex operating system in C. The absence of restrictions and the generality of C can make it more powerful than many so-called "very-high-level" languages for some applications.

UNIX Pipes

UNIX commands are simply executable program files. Files are executed by naming them to a command-loading program called the shell. An executable file is produced by the loader, which combines (links) object programs together to form an executable result. These object programs are the results of earlier compilation or assembly. Object programs are produced by the various

available compilers such as C, FORTRAN 77, and PASCAL. Object programs may be kept in a user's directory on object files, or they may be collected together into object program libraries.

Under UNIX, every program is associated with three input/output (i/o) streams, called stdin, stdout, and stderr, for standard input, standard output, and standard error, respectively (see Figure 1).

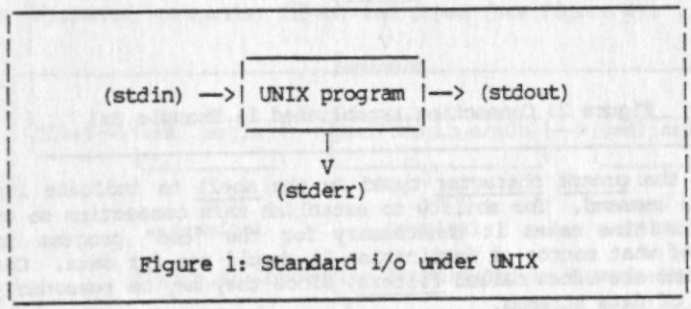


Figure 1: Standard i/o under UNIX

When a UNIX program is executed, the shell connects all three of these i/o streams to the terminal unless prevented from doing so by the user or the program. Typical UNIX programs read input data from stdin, generate output on stdout, and post any necessary error messages on stderr. Of course, programs may also read and write files which are not associated with stdin, stdout, or stderr, but the standard i/o connections often suffice.

The advantage of the UNIX standard i/o configuration is that it allows most programs to operate independently of the source or destination of the data they process. This is possible because all files are treated identically by UNIX: as streams of data bytes. Individual bytes may represent characters, pairs of bytes may represent 16-bit integers, quartets of bytes may represent 32-bit floating point or integer data, octets of bytes may represent 64-bit double-precision floating point data, etc. UNIX always treats byte streams identically for purposes of storage or transmission.

When a program is executed by naming it to the shell, the user may specify connections for stdin, stdout, and stderr in two basic ways: by redirecting the i/o streams to files, or by joining the standard output of one program with the standard input of another via the "pipe" mechanism.

For example, the notation

```
%cmd <file1 >file2                    (Ex1)
```

indicates to the shell that the program on file "cmd" is to be loaded into memory and executed with its standard input connected to "file1" and its standard output connected to "file2" (see Figure 2).

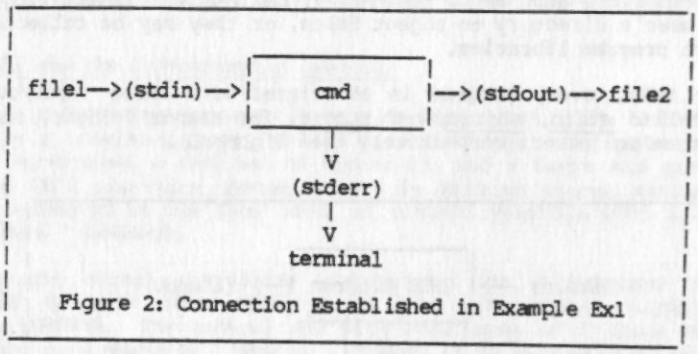


Figure 2: Connection Established in Example Ex1

The "%" is the prompt character typed by the shell to indicate its readiness to accept a command. The ability to establish this connection so conveniently at execution time makes it unnecessary for the "cmd" program to have any knowledge of what source or destination it should use for data. Consequently, UNIX programs are often called filters, since they may be reasonably viewed as processors of data streams.

The utility of stderr is clear. If the standard output is connected to some place besides the user's terminal, it is still convenient to have a connection to the terminal for error messages, termination notices, etc.

The UNIX notation

```
%cmd1 | cmd2                                    (Ex2)
```

specifies that the standard output of program "cmd1" is to be connected to the standard input of "cmd2", and the two programs are to be run concurrently (i.e., simultaneously in a timeshared fashion). The standard input of "cmd1" and the standard output of "cmd2" are left connected (simultaneously!) to the user's terminal in this example (see Figure 3). The standard i/o connections for the terminals are left implicit in the figure, i.e., "cmd1" is connected to the terminal via both stdin and stderr, etc.

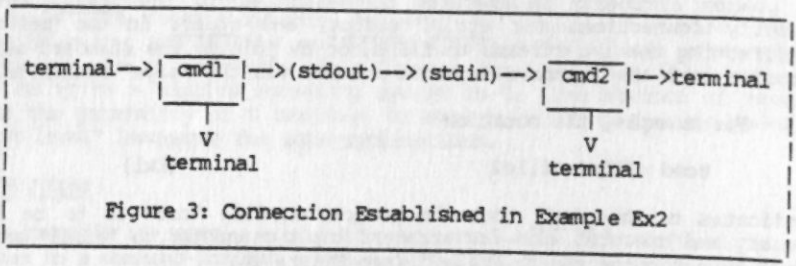


Figure 3: Connection Established in Example Ex2

This connection is established by the shell, and consists of a "hidden" buffered data path called a pipe between the two processes. The synchronization of the two processes is automatically handled by UNIX, with "cmd2" waiting

until "cmd1" has produced some output before it reads input from the pipe. On the Berkeley-modified UNIX/32V virtual operating system used at CARL, pipes buffer up to 4096 bytes of data between processes.

Pipes and file redirections of standard i/o may be combined. For example, the command line

```
%cmd1 arg1a <file1 | cmd2 arg1b arg2b | cmd3 arg1c >file2    (Ex3)
```

combines arguments, commands, files, and pipes (see Figure 4).

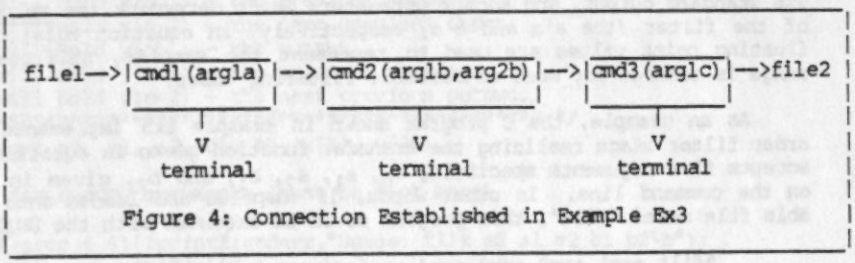


Figure 4: Connection Established in Example Ex3

Each program named may access its own arguments (if any), but not the arguments of other programs in a piped command such as this one. "cmd1" is given one argument. It has its standard input connected to file1, and its standard output connected to the standard input of "cmd2" with a pipe. "cmd2" has two arguments, and processes the data which is output by "cmd1". The output of "cmd2" is passed to "cmd3", which has one argument and places its results on file2. All three programs run concurrently and are automatically synchronized by UNIX.

6.5. Cascade Filter Connections via UNIX Pipes

The resemblance between pipe connections and cascade filter connections is rather evident. If the data passed between the programs represent digital signals, and the programs represent signal processing algorithms, UNIX provides a convenient and flexible way to provide arbitrary cascade connections between these programs.

Suppose, for example, that we implement a general second-order digital filtering stage as a UNIX C program. We are assured by digital signal processing theory that any digital filter can be decomposed into some number of second-order stages, since

$$H(z) = \frac{Y(z)}{X(z)} = \frac{\sum_{i=0}^{N} a_i z^{-i}}{\sum_{i=0}^{N} b_i z^{-i}} \qquad \text{(Eq1a)}$$

where b_0 is defined as 1, may be rewritten in the form

$$H(z) = \frac{Y(z)}{X(z)} = g_0 H_1(z) H_2(z) \ldots H_K(z) \qquad \text{(Eq1b)}$$

where K is the integer part of (N+1)/2, and $H_i(z)$ is a second-order section, i.e.,

$$H_i(z) = \frac{a_{0_i} + a_{1_i} z^{-1} + a_{2_i} z^{-2}}{1 + b_{1_i} z^{-1} + b_{2_i} z^{-2}} \qquad \text{(Eq1c)}$$

A completely general implementation of this second-order filter as a C program would simply read samples from its standard input, write samples on its standard output, and accept parameters which determine the zeros and poles of the filter (the a's and b's, respectively, in equation Eq1a). If 32-bit floating point values are used to represent the samples, sufficient dynamic range is assured for most conceivable filtering cases.

As an example, the C program shown in Example Ex5 implements a second-order filter stage realizing the transfer function given in equation Eq1c. It accepts five arguments specifying a_0, a_1, a_2, b_1, and b_2, given in that order on the command line. In other words, if compiled and loaded onto an executable file named "filt", this program could be executed with the UNIX command:

```
%filt arg1 arg2 arg3 arg4 arg5 <input >output          (Ex4)
```

where arg1 is a numerical value for a_0, arg2 a value for a_1, etc., input data comes from file "input", and output data is stored on file "output".

```
/* cc <thisfile.c> -o filt */

#include <stdio.h>

main(argc, argv)
 int argc; char *argv[];
{
/* ************************************************
x[0] will hold x(n)   - the current input,
x[1] will hold x(n-1) - the previous input,
x[2] will hold x(n-2) - the next previous input,
y[0] will hold y(n)   - the output,
y[1] will hold y(n-1) - the previous output,
y[2] will hold y(n-2) - the next previous output.
************************************************ */
    float a0, a1, a2, b1, b2, x[3], y[3];
/*
Check for enough arguments (must be 5 or more)
*/
    if(argc < 6){fprintf(stderr,"Usage: filt a0 a1 a2 b1 b2\n");
                 exit(-1); }
/*
Collect filter coefficients (first argument specifies a0, etc.)
*/
    a0 = atof(argv[1]); a1 = atof(argv[2]); a2 = atof(argv[3]);
                        b1 = atof(argv[4]); b2 = atof(argv[5]);
/*
MAIN COMPUTING LOOP: Read floating point sample values from
standard input into x[0], until end of input data is reached
*/
    while( fread(&x[0], sizeof(float), 1, stdin) ){
/*
Compute output sample as weighted combination of input,
past inputs, and past outputs
*/
     y[0] = a0*x[0] + a1*x[1] + a2*x[2] - b1*y[1] - b2*y[2];
/*
Filter output goes to standard output
*/
     fwrite(&y[0], sizeof(float), 1, stdout);
/*
Rotate previous inputs and outputs for next pass through loop
*/
     x[2] = x[1]; x[1] = x[0]; y[2] = y[1]; y[1] = y[0];
    }
}
```

Example Ex5: A General-Purpose Second-Order Filter C Program (See Text)

A three-stage cascade realization of the transfer function

$$H(z) = \frac{1+z+z^{-2}}{(1 - \frac{1}{4}z^{-1} - \frac{1}{8}z^{-2})(1 - \frac{1}{3}z^{-1})(1 + \frac{1}{2}z^{-1} + \frac{1}{2}z^{-2})} \qquad \text{(Eq2)}$$

could then be implemented by the UNIX command

```
%filt 1 1 1 -.25 -.125 <input | filt 1 0 0 -.33 0
                             | filt 1 0 0 .5 .5 >output        (Ex6)
```

Notice that the second stage in this cascade realization effectively uses the second-order filt program as a first-order section, since both a_2 and b_2 are set to zero. Some wasted multiplication could be saved by writing a separate first-order section program similar to the second-order program given above. This program would only require three arguments in order to realize the transfer function

$$H(z) = \frac{a_0+a_1z^{-1}}{1+b_1z^{-1}} \qquad \text{(Eq3)}$$

6.6. Parallel Filter Connections Using UNIX Pipes: The "para" Program

Another important method of realizing digital filters uses parallel connections. This type of realization is based .. the use of partial fraction expansion to express the transfer function in the form

$$H(z) = C + \sum_{i=1}^{K} H_i(z) \qquad \text{(Eq4)}$$

where each $H_i(z)$ is either a second-order section or a first-order section similar to the forms given above. For example, the transfer function

$$H(z) = \frac{3 + \frac{5}{3}z^{-1} + \frac{2}{3}z^{-2}}{(1 - \frac{1}{3}z^{-1})(1 + \frac{1}{2}z^{-1} + \frac{1}{2}z^{-2})} \qquad \text{(Eq5a)}$$

may be expressed in the form

$$H(z) = \frac{2}{1 - \frac{1}{3}z^{-1}} + \frac{1+z^{-1}}{1 + \frac{1}{2}z^{-1} + \frac{1}{2}z^{-2}} = H_1(z) + H_2(z) \qquad \text{(Eq5b)}$$

To realize this filter in UNIX, we must somehow arrange for the input signal to be applied simultaneously to both of the parallel stages corresponding to $H_1(z)$ and $H_2(z)$. Furthermore, we must also arrange for the output to be the sum of the outputs of these two parallel stages.

There is no direct, convenient way to accomplish this using pipes alone. However, a special C program can allow such structures to be implemented.

The program used at CARL for this purpose is called "para". "para"

provides parallel signal processing by feeding its standard input(through pipes) to each of several processing pipelines which run in parallel, summing the outputs of these processes, and writing the resulting summed data stream on the standard output.

For example, the statement

```
%para 2 "filt1 ..." "filt2 ..." <input >output        (Ex7)
```

applies the "filt1" and "filt2" commands (with their respective arguments, represented by the ellipses here) to two separate copies of the signal input, then adds the parallel output signals from these filters and writes their sum on the standard output of the "para" process. The number 2 in the command specifies the number of parallel channels to be used.

If only one process is specified for multiple channels, it is applied in parallel to each channel.

A diagrammatic representation of this filter structure is shown in Figure 5.

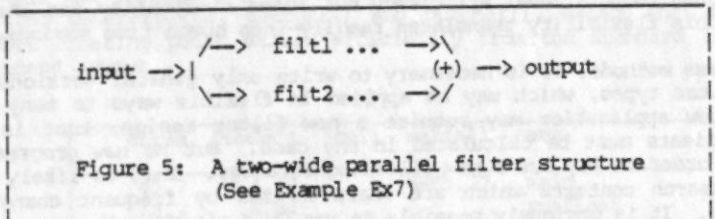


Figure 5: A two-wide parallel filter structure
(See Example Ex7)

If only one process is specified, it is applied in each parallel process. The structure of

```
%para 4 "filt ..." <input >output                    (Ex8)
```

is represented in Figure 6.

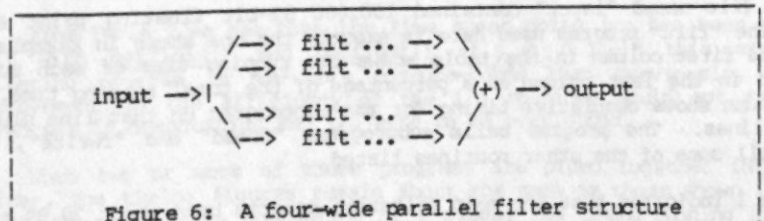


Figure 6: A four-wide parallel filter structure
(See Example Ex8)

The parallel processes must be enclosed in quote (") marks, and may themselves be pipes, such as

```
%para 4 "filt ... | filt ..." <input >output                (Ex9)
```

Using the "para" program, we can write a UNIX command which will realize the filter specified in equation Eq5b:

```
%para 2 "filt 2 0 0 -.33 0"
                "filt 1 1 0 .5 .5" <input >output          (Ex10)
```

Obviously, combinations of parallel and cascade filter realizations are possible using the approaches given above.

6.7. Efficiency

The efficiency of these techniques for realizing filters must be less than that of writing special programs for each case. The advantage gained here is not added computational efficiency but added flexibility for the computer user. This flexibility translates readily into human time savings.

Using these methods, it is necessary to write only general versions of a few basic filter types, which may be applied in flexible ways to many problems. Each new application may require a new filter design, that is, the filter coefficients must be calculated in any case. But no new program need be written in order to try out a new filtering strategy. This is likely to be useful in research contexts which are characterized by frequent changes in filter designs. It is obviously possible to use UNIX piping methods to design and evaluate satisfactory filters which can then be optimized as specially-written computer programs, if they are needed for high-volume production applications.

Even so, the absolute efficiency of programs such as the second-order filter example given above is not bad. Table 1 shows the UNIX profile results of running the command

```
%filt 1 0 0 0 0 <input >output                             (Ex11)
```

where the file named "input" contained 100,000 32-bit floating point sample values. The "filt" program used here is exactly the one shown in Example Ex5, above. The first column in the table shows the running time of each subprogram named in the last column as a percentage of the total running time. The second column shows cumulative timing for the subprogram on that line plus all previous lines. The program calls subprograms "fread" and "fwrite", which further call some of the other routines listed.

Table 1 indicates that the main filtering program used about 29.9% of the total running time for computation, and that the i/o routines used a total of 29.2 + 27 = 56.2% of the total running time. If we define the time spent doing computation as t_c and the time spent doing i/o as $t_{i/o}$, then we have $t_{i/o}/t_c \simeq 2$ in this example. Therefore, speeding up the i/o processes would have an effect on the throughput time of the program, since i/o and

computation are normally overlapped (among different processes) in UNIX.

%time	cumsecs	#call	ms/call	name
29.9	6.26	1	6255.71	_main
29.2	12.37			_fread
27.0	18.02			_fwrite
7.0	19.49			_write
4.9	20.50			_read
2.0	20.91			mcount
0.2	20.96			__filbuf

Table 1: Profile Results of Running Second-Order Filter Program on 100,000 Data Points

Table 2 shows UNIX profile results from another run of the same program on the same data, but with improved i/o routines written at CARL, called getfloat() and putfloat(). These routines work like their character-reading and writing counterparts, getchar() and putchar(), except that they get and put 32-bit floating point values efficiently from the standard input and to the standard output.

%time	cumsecs	#call	ms/call	name
40.7	4.69	1	4685.27	_main
18.7	6.84			_putfloat
17.0	8.79			_getfloat
12.2	10.19			_write
10.6	11.40			_read
0.4	11.45			_fgetfloat
0.4	11.50			_flushfloat

Table 2: Profile Results of Running Improved Second-Order Filter Program on 100,000 Data Points

From Table 2 we see that the time spent doing i/o has been cut about in half with getfloat() and putfloat() ($t_{i/o}/t_c \simeq 1$ in this example). The filtering program is now more compute-bound than previously, and further improvement in the efficiency of the i/o routines would not result in any advantage if computation and i/o were to be overlapped.

When two or more of these programs are piped together into a cascade filter, the timing figures remain about the same as those shown in the tables above. For a cascade connection of P pipes, the total running time is given by

$$T_{piped} = P(t_{i/o} + t_c) \quad . \quad .6)$$

A special program written to perform the identical P-stage filtering computation would still have to do i/o, but only once, so its running time would be

$$T_{program} = t_{i/o} + Pt_c \qquad \text{(Eq7)}$$

The throughput time saved by writing a special program is therefore equal to

$$T_{diff} = T_{piped} - T_{program} = t_{i/o}(P - 1) \qquad \text{(Eq8)}$$

where P is the number of cascade stages in the filter and $t_{i/o}$ is the time spent doing i/o by any process for the given amount of data.

We can define

$$\text{Penalty Ratio} = \frac{t_{piped}}{t_{program}} = \frac{P(t_{i/o} + t_c)}{t_{i/o} + Pt_c} = \frac{P(\phi + 1)}{P + \phi} \qquad \text{(Eq9)}$$

where $\phi = t_{i/o}/t_c$. The Penalty Ratio gives a rough measure of the throughput penalty associated with the UNIX pipe mechanism. For the data given in Table 2 above, where $t_{i/o}$ is approximately equal to t_c and P = 3, the relative penalty paid for using the pipe method is a factor of about 1.5. In other words, the throughput time for the piped version is about 50% longer than a specially written program. Of course, a specially-written program could be optimized around the filter structure being realized, yielding greater savings than that indicated above. However, this seems not too inordinate a price to pay to avoid having to write a new program for many cases.

6.8. Multichannel Signal Processing: The "chan" Program

An important option for audio signals is the use of multiple channels to represent spatial aspects of sound. A multichannel audio signal is usually represented in digital form as a time-division multiplexed sequence of numerical sample values, with each consecutive group of N samples giving values for the N audio channels. To process such multiplexed signals, they must be first demultiplexed, then processed channel-by-channel, then remultiplexed after processing.

The program used at CARL for this purpose is called "chan". "chan" provides multichannel signal processing by demultiplexing its standard input, feeding each channel (through pipes) to one of several monophonic processing pipelines which run as parallel processes, remultiplexing the outputs of these processes, and writing the resulting multiplexed data stream on the standard output. A typical command of the form

```
%chan 2 "filtl ..." "filt2 ..." <stereoin >stereoout        (Ex12)
```

takes data from file "stereoin", separates the 2 channels, applies "filtl ..." to the first channel, and "filt2 ..." to the second, then recombines the sig-

nal and writes the results on file "stereoout".

The above example is shown diagrammatically in Figure 7.

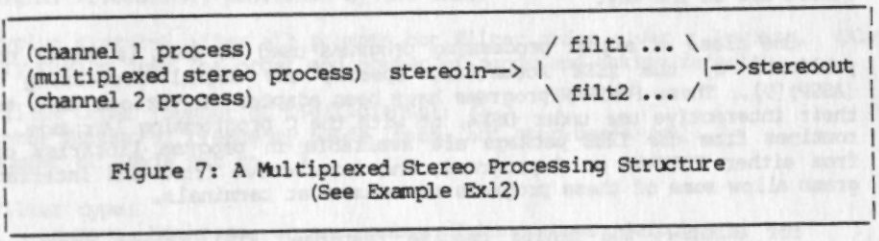


Figure 7: A Multiplexed Stereo Processing Structure
(See Example Ex12)

If only one process is specified for multiple channels, it is applied in parallel to each channel. For example, the command

```
%chan 4 "filt ..." <quadin >quadout                    (Ex13)
```

is shown in Figure 8.

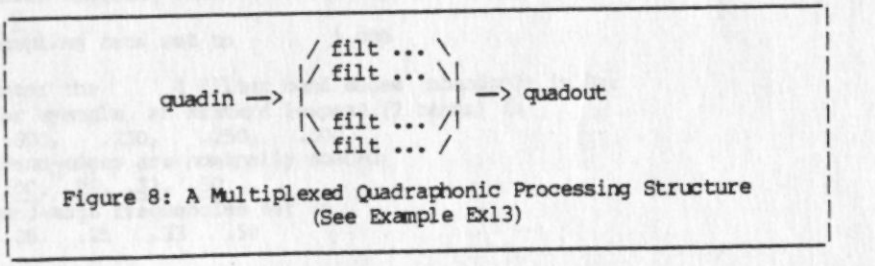


Figure 8: A Multiplexed Quadraphonic Processing Structure
(See Example Ex13)

The channel processes must be enclosed in quote (") marks, and may themselves be pipes, such as

```
%chan 4 "filt ... | filt ... " <in >out          (Ex14)
```

6.9. The UNIX Signal Processing Environment

The efficiency considerations discussed with relation to the "para" program also apply to the "chan" program. However, the "chan" program provides an even more compelling reason to minimize our concern with overall efficiency: "chan" makes it possible to keep only one, simple, general, single channel version of any processing program, such as a filtering program, and to apply it to an arbitrary number of multiplexed signal channels. No new programs need to be written to go from mono to stereo, or from stereo to quad.

This makes it possible to consider writing a basic set of signal processing programs for general use, since any of these programs may be replicated

and run in concurrent fashion through the mechanism of the UNIX pipe, working with the "para" and "chan" programs. Such a set of signal processing programs is now under development at CARL. Several useful programs exist, and several others are on the way.

One class of signal processing programs used at CARL is based on those published by the IEEE Acoustics, Speech, and Signal Processing Society (ASSP)[9]. These FORTRAN programs have been adapted in various ways to allow their interactive use under UNIX, or with the C programming language. Useful routines from the IEEE package are available in program libraries callable from either FORTRAN or C programs, and interactive front-end interface programs allow some of these programs to be used at terminals.

For example, the finite impulse response (FIR) filter design program available from the IEEE has been given a conversational front-end, allowing a relatively unsophisticated user to design and use effective FIR filters. The following sample dialogue illustrates the interactive design of a 31-tap lowpass FIR filter. The responses of the user are underlined.

Finite Impulse Response (FIR) digital filter design program. This program is fully explained with examples in chapter five of Programs for Digital Signal Processing, published by the IEEE.

Typing <return> after all prompts but filter order gives a lowpass. (Also try giving just the order and number of bands and taking defaults)

```
Filter Order (number of coefficients): 31
Enter number of frequency bands (pass- and stop-bands)(=2): 2
Number of bands set to   2

Filter type:
1 - Lowpass, highpass, or any number of bandpasses (default)
2 - Differentiator
3 - Hilbert transformer
1
Type 1 selected

Praise Fortran! You must now include DECIMAL POINTS in all remaining
values.  Also, use COMMAS to separate numbers

Enter sampling rate in Hz (default is 1.0)
1.0
Sampling rate set to          1.000

Enter the       4 filter band edges (nbands*2) in Hz:
For example, an srate/4 lowpass (2 bands) is
 .000,   .230,   .250,   .500,
(Band-edges are nominally spaced)
 .00, .25, .33, .50
Band-edge frequencies set to
 .00   .25   .33   .50

Enter desired function across the   2  bands
(e.g. 0,1.=highpass, 1.,0=lowpass, 0,1.,0=highpass)
<return>
Filter Band Gains set to
1.0    .0

Enter desired relative weighting across bands:
<return>
(Error ripple in 1st band will be 10 times larger than rest)
Band weighting set to
1.0  10.0
```

Example Ex15a: The CARL "fir" Program Dialogue (See Text)

The program then computes the specified filter, and produces the following report on the user's terminal:

```
**************************************************************************

                finite impulse response (fir)
              linear phase digital filter design
                 remez exchange algorithm

                      bandpass filter

                    filter length =  31

                  ***** impulse response *****
                 h( 1) =   .11952063e-02 = h( 31)
                 h( 2) =  -.40142922e-02 = h( 30)
                 h( 3) =  -.77995238e-02 = h( 29)
                 h( 4) =   .43392329e-02 = h( 28)
                 h( 5) =   .84881716e-02 = h( 27)
                 h( 6) =  -.12557141e-01 = h( 26)
                 h( 7) =  -.77216978e-02 = h( 25)
                 h( 8) =   .24788539e-01 = h( 24)
                 h( 9) =  -.75214682e-03 = h( 23)
                 h(10) =  -.40120058e-01 = h( 22)
                 h(11) =   .23747315e-01 = h( 21)
                 h(12) =   .55469174e-01 = h( 20)
                 h(13) =  -.78259802e-01 = h( 19)
                 h(14) =  -.66905727e-01 = h( 18)
                 h(15) =   .30829692e+00 = h( 17)
                 h(16) =   .57114248e+00 = h( 16)

                          band  1         band  2
lower band edge            .0000000        .3333333
upper band edge            .2500000        .5000000
desired value             1.0000000        .0000000
weighting                 1.0000000      10.0000000
deviation                  .0124692        .0012469
deviation in db            .1076362     -58.0832291

extremal frequencies--maxima of the error curve
    .0000000    .0371094    .0742188    .1093750    .1445313
    .1796875    .2128906    .2382813    .2500000    .3333333
    .3411458    .3587240    .3821614    .4095052    .4388021
    .4700521    .5000000
```

Example Ex15b: The CARL "fir" Program Dialogue (See Text)

The user is then asked to specify a name for a file to hold the filter specifications, called a filterfile.

```
Output filename: test.flt
Writing: test.flt
```

Example Ex15c: The CARL "fir" Program Dialogue (See Text)

The filterfile, called "test.flt", may be used as an argument to a general-purpose filtering program called "filter". This program can be used to filter samples produced by the "cmusic" sound synthesis program, for example, via UNIX pipes with the command

```
%cmusic scorefile | filter test.flt >outputfile      (Ex16)
```

where "scorefile" is the name of a file containing specifications for the "cmusic" program, and the filtered results are stored on file "outputfile".

Of course the "filter" program could be used with the "para" or "chan" programs described earlier.

Other programs for signal processing based on the IEEE package currently available at CARL include "srconv", a program for changing the sampling rate of a digital signal, "lpc", a program for linear predictive coding (LPC) of a digital signal, and "spect", a program for obtaining the frequency spectrum of a digital signal via the fast Fourier transform (FFT) algorithm. In addition, additional signal processing utility programs have been written to obtain the peak or rms amplitude values of a digital signal, to scale the signal by a factor, to delay a signal, etc. A signal-processing "template" program called "wire.c", with transfer function $H(z) = 1$, is available for easy modification to other transfer functions. Finally, all program documentation can be conveniently kept on-line, allowing users to keep abreast in a dynamically-changing programming environment. Figure 9 shows a sample display from the "hist" program for calculating a histogram of amplitude values found in a digital signal.

```
| >  1.00000 (=   0.0dB) =    0 |
| >  0.50000 (=  -6.0dB) =    0 |
| >  0.25000 (= -12.0dB) = 1209 |*****
| >  0.12500 (= -18.1dB) = 5500 |************************
| >  0.06250 (= -24.1dB) = 7971 |***********************************
| >  0.03125 (= -30.1dB) = 8964 |****************************************
| >  0.01563 (= -36.1dB) = 7041 |*******************************
| >  0.00781 (= -42.1dB) = 5262 |***********************
| >  0.00391 (= -48.2dB) = 3281 |**************
| >  0.00195 (= -54.2dB) = 2083 |*********
| >  0.00098 (= -60.2dB) = 1186 |*****
| >  0.00049 (= -66.2dB) =  662 |**
| >  0.00024 (= -72.2dB) =  391 |*
| >  0.00012 (= -78.3dB) =  177 |
| >  0.00006 (= -84.3dB) =   98 |
| > -0.00000 (=-120.0dB) = 5327 |***********************
| Mean = 0.000021 (= -93.7dB)
| Peak = -0.462264 (= -6.7dB)
| RMS = 0.089411 (= -21.0dB)
| 49152 samples
|
| Figure 9: Histogram Display Produced by "hist" Program.
| (There were 1,209 samples between .25 and .5, 5,500 samples in the
| range .125 to .25, etc.  The mean, peak, and rms amplitude values
| are included at the bottom of the display, along with a count of
| the number of samples.
```

6.10. Conclusion

The UNIX operating environment can be a flexible tool for digital signal processing applications. The pipe facility can be combined with special programs which allow cascade and parallel connections among signal processing program modules to be formed in a convenient and flexible way. The multichannel processing facility can be invoked to allow any signal processing program to be applied separately to individual channels of a multiplexed data stream. Interactive programs for filter design and signal measurement can be added to this context easily, making this UNIX operating environment pleasant for signal processing applications.

6.11. Acknowledgements

I wish to acknowledge the helpful contributions of Julius Smith, who wrote some of the original interactive interface programs for the IEEE software, and the vision of the original designers of UNIX, Ken Thompson and Dennis Ritchie of Bell Laboratories, who years ago forged an incredibly powerful tool for computing whose flexibility is still being discovered.

```
/*
* para provides parallel signal processing by feeding its stdin (through pipes)
* to each of several monophonic processing pipelines which run in parallel,
* summing the outputs of these processes, and writing the resulting
* summed data stream on the standard output. A typical command form:
*
* cmusic score.sc | para 2 "reverb .6 1000" "reverb .7 1131" | sndout
*
* applies both "reverb .6 1000" and "reverb .7 1131" to separate copies
* of the output of cmusic, then adds the parallel signals and pipes
* the sum to sndout.  If only one process is specified for multiple channels,
* it is applied in parallel to each channel.
*                                                          -frm
*/

#include <stdio.h>

#define MAXCHAN 4
#define ERR(msg) {fprintf(stderr,msg); exit(-1);}
#define MS(msg,val) {fprintf(stderr,msg,val);}

main(argc, argv) int argc; char *argv[]; {
    register i, nchan;
    int fdi[MAXCHAN][2], fdo[MAXCHAN][2];
    FILE *pipe_in[MAXCHAN], *pipe_out[MAXCHAN];
    float sample;
    int data, status;

    if(argc > 1) nchan = atoi(argv[1]);
    if (argc != 3 && argc != nchan + 2)
        ERR("Usage: para N `cmd' -or- para N `cmd1' ... `cmdN'\n");
    if(nchan < 2 || nchan > MAXCHAN)
        ERR("para: illegal number of channels\n");

    for(i = 0; i < nchan; i++)
        if( pipe( &fdi[i][0] ) < 0 || pipe( &fdo[i][0] ) < 0)
            ERR("para: pipe overflow\n");
/*
   spouse process which sums output pipes, and writes on stdout
*/
    if( !fork() ){
        for(i = 0; i < nchan; i++){
            close( fdi[i][0] );
            close( fdi[i][1] );
            close( fdo[i][1] );
            pipe_out[i] = fdopen( fdo[i][0], "r");
        }

        if( !isatty(1)) while(1){
         register float sum;
            for(sum = 0.0, i = 0; i < nchan; i++, sum += sample)
```

```
                data = fread( &sample, sizeof(float), 1, pipe_out[i] );
            if (data) fwrite( &sum, sizeof(float), 1, stdout );
                else exit(0);
        } else while(1){
         register float sum;
            for(sum = 0.0, i = 0; i < nchan; i++, sum += sample)
                data = fread( &sample, sizeof(float), 1, pipe_out[i] );
            if (data) printf("%f\n",sum);
                else exit(0);
        }
    }
/*
   processing children read stdin and write stdout, which are connected
   to process input and process output pipes, respectively
*/
    for(i = 0; i < nchan; i++)if( !fork() ){
    /* connect stdin to process input pipes */
        close( fdi[i][1] );
        dup2( fdi[i][0], 0 );
        close( fdi[i][0] );
    /* connect stdout to process output pipes */
        close( fdo[i][0] );
        dup2( fdo[i][1], 1 );
        close( fdo[i][1] );
    /* execute processes */
        if(argc == 3)
            execlp( "/bin/csh", "csh", "-c", argv[2], 0);
        else
            execlp( "/bin/csh", "csh", "-c", argv[i+2], 0);

        ERR("para: We NEVER come here!");
    }
/*
   parent reads its stdin, and writes on input pipes of each process
*/
    for(i = 0; i < nchan; i++){
        close( fdo[i][0] );
        close( fdo[i][1] );
        close( fdi[i][0] );
        if( (pipe_in[i] = fdopen( fdi[i][1], "w") ) == NULL)
            ERR("para: fdopen failed\n");
    }

    if( !isatty(0) ) while( fread( &sample, sizeof(float), 1, stdin ) ){
        for( i = 0; i < nchan; i++ )
            if( !fwrite( &sample, sizeof(float), 1, pipe_in[i] ) )
                ERR("para: unable to write process input pipes\n");
    } else wait( &status );
}
```

```
/*
* chan provides multichannel signal processing by demultiplexing its standard
* input, feeding each channel (through pipes) to one of several monophonic
* processing pipelines which run as parallel processes, remultiplexing the
* outputs of these processes, and writing the resulting multiplexed data stream
* on the standard output. A typical command form:
*
* cmusic stereo.sc | chan 2 "reverb .6 1000" "reverb .7 1131" | sndout -c 2
*
* takes the output of cmusic, separates the 2 channels, applies
* "reverb .6 1000" to the first channel, and "reverb .7 1131" to the second,
* then recombines the signal and pipes it to sndout.  If only one process
* is specified for multiple channels, it is applied in parallel to each channel.
*                                                          -frm
*/

#include <stdio.h>

#define MAXCHAN 4
#define ERR(msg) {fprintf(stderr,msg); exit(-1);}
#define MS(msg,val) {fprintf(stderr,msg,val);}

main(argc, argv) int argc; char *argv[]; {
    register i, nchan;
    int fdi[MAXCHAN][2], fdo[MAXCHAN][2];
    FILE *pipe_in[MAXCHAN], *pipe_out[MAXCHAN];
    float frame[MAXCHAN];
    int data, status;

    if(argc > 1) nchan = atoi(argv[1]);
    if (argc != 3 && argc != nchan + 2)
        ERR("Usage: chan N `cmd' -or- chan N `cmd1' ... `cmdN'\n");
    if(nchan < 2 || nchan > MAXCHAN)
        ERR("chan: illegal number of channels\n");

    for(i = 0; i < nchan; i++)
        if( pipe( &fdi[i][0] ) < 0 || pipe( &fdo[i][0] ) < 0)
            ERR("chan: pipe overflow\n");
/*
   spouse process which reads output pipes, remuxes, and writes on stdout
*/
    if( !fork() ){
        for(i = 0; i < nchan; i++){
            close( fdi[i][0] );
            close( fdi[i][1] );
            close( fdo[i][1] );
            pipe_out[i] = fdopen( fdo[i][0], "r");
        }

        if( !isatty(1) )while(1){
            for(i = 0; i < nchan; i++)
             data = fread( &frame[i], sizeof(float), 1, pipe_out[i] );
            if (data) fwrite( &frame[0], sizeof(float), nchan, stdout );
                else exit(0);
```

```
        } else while(1){
            for(i = 0; i < nchan; i++)
             data = fread( &frame[i], sizeof(float), 1, pipe_out[i] );
            if (data) for(i = 0; i < nchan; i++) printf("%f\n", frame[i]);
                else exit(0);
        }
    }
/*
   processing children read stdin and write stdout, which are connected
   to process input and process output pipes, respectively
*/
    for(i = 0; i < nchan; i++)if( !fork() ){
    /* connect stdin to process input pipes */
        close( fdi[i][1] );
        dup2( fdi[i][0], 0 );
        close( fdi[i][0] );
    /* connect stdout to process output pipes */
        close( fdo[i][0] );
        dup2( fdo[i][1], 1 );
        close( fdo[i][1] );
    /* execute processes */
        if(argc == 3)
            execlp( "/bin/csh", "csh", "-c", argv[2], 0);
        else
            execlp( "/bin/csh", "csh", "-c", argv[i+2], 0);

        ERR("chan: We NEVER come here!");
    }
/*
   parent process reads its stdin, demuxes, and writes on input pipes
   of each parallel process
*/
    for(i = 0; i < nchan; i++){
        close( fdo[i][0] );
        close( fdo[i][1] );
        close( fdi[i][0] );
        if( (pipe_in[i] = fdopen( fdi[i][1], "w") ) == NULL)
            ERR("chan: fdopen failed\n");
    }

    if( !isatty(0) ) while( fread( &frame[0], sizeof(float), nchan, stdin ) ){
        for( i = 0; i < nchan; i++ )
            if( !fwrite( &frame[i], sizeof(float), 1, pipe_in[i] ) )
                ERR("chan: unable to write process input pipes\n");
    } else wait( &status );
}
```

- REFERENCES -

1. J. W. Cooley and J. W. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," Math. Computation, Vol. 19, 1965, pp. 297-301.

2. T. A. Dolotta, R. C. Haight, and J. R. Mashey, "The Programmer's Workbench - A Machine for Software Development," Comm. ACM, Vol. 20, No. 10, Oct. 1977, pp. 746-753.

3. Brian W. Kernighan and John R. Mashey, "The UNIX Programming Environment," Computer, Vol. 14, No. 4, April 1981, pp. 12-24.

4. Brian W. Kernighan and Dennis M. Ritchie, The C Programming Language, Prentiss-Hall, Inc., 1978.

5. Harry Y-F. Lam, Analog and Digital Filters: Design and Realization, Prentiss-Hall, Inc., 1979.

6. F. Richard Moore, "The FRMbox: A Modular, Digital, Realtime Music Synthesizer," Computer Music J., (in press).

7. James A. Moorer, "Synthesizers I have Known and Loved," Computer Music J., Vol. 5, No. 1, Spring 1981, pp. 4-12.

8. Alan V. Oppenheim and Ronald W. Schafer, Digital Signal Processing, Prentiss-Hall, Inc., 1975.

9. Programs for Digital Signal Processing, Edited by the Digital Signal Processing Committee, IEEE Acoustics, Speech, and Signal Processing Society, IEEE Press, 1979.

10. Lawrence R. Rabiner and Bernard Gold, Theory and Application of Digital Signal Processing, Prentiss-Hall, Inc., 1975.

11. D. M. Ritchie, S. C. Johnson, M. E. Lesk, and B. W. Kernighan, "UNIX Time-Sharing System: The C Programming Language," Bell System Technical J., Vol. 57, No. 6, Oct. 1978, pp. 1991-2019.

12. D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," Comm. ACM, Vol. 17, No. 7, July 1974, pp. 365-375.

13. M. J. Rochkind, "The Source Code Control System," IEEE Trans. Software Eng., Vol. SE-1, No. 4, Dec. 1975, pp. 364-370.

NAME

asw – simple audio switch patcher

SYNOPSIS

asw [**-d**] [**-** *coutput_device*] [**-r**] [**-R**"*raw asw command*"] [**-i**]

DESCRIPTION

asw is a simple program to manually make or break audio switch connections via the audio switch daemon. It also allows one to send a reset command to the audio switch daemon, and to inform the daemon to re-read the configuration file.

The **-d** flag disconnects all output devices that are connected to your input space (determined by your tty).

The **-c** flag connects the specified *output_device* to your input space.

The **-r** flag issues a reset command to the switch. All connections are broken. This command should only be used if the audio switch or the daemon has become confused.

The **-R** flag is for handing the audio switch daemon a raw command. The command should be enclosed in quotes to protect the spaces in it. The format of the command is explained in the paper *The CARL Audio Switch Daemon*.

The **-i** flag is for re-initializing the audio switch daemon. It causes the daemon to re-read the configuration file. This obviates the need to figure out the process id of the daemon and send it a SIGHUP command after the configuration file has been changed.

The possible *output_devices* (and *input_spaces*) for a particular installation are specified in the configuration file.

FILES

/usr/local/lib/asw-conf

BUGS

The **-R** and **-r** flags are backwards. One might infer that **-R** is a more important command and care should be excercised when using it since it is upper case, and that **-r** requires less care when just the opposite is true.

AUTHOR

Rusty Wright

SEE ALSO

aswdaemon(C), *The CARL Audio Switch Daemon*

NAME

aswdaemon – audio switch daemon

SYNOPSIS

asw [**– d**] [**– c***config_file*]

DESCRIPTION

The **aswdaemon** is a daemon process that manages the audio switch. **aswdaemon** is completely described in the paper ***The CARL Audio Switch Daemon*** which is is in the doc directory of the CARL distribution. All that is described here is the flags.

The **– d** flag enables a very simple form of debugging that allows the daemon to use an alternate socket.

The **– c** flag is used to specify an alternate configuration file.

AUTHOR

Rusty Wright

SEE ALSO

asw(C)

NAME

atob - arabic to binary converter for floatsams

SYNOPSIS

atob [**file**] < arabic numbers > floatsams

DESCRIPTION

atob reads arabic numbers, converts them to floatsams (32-bit binary floating point samples) and writes them on its standard output.

If **file** is named, it is expected to contain free field arabic numbers. Otherwise, **atob** reads its standard input. Input conversion is done with **fscanf(3)**.

AUTHOR

Gareth Loy

SEE ALSO

btoa(1carl), xform(1carl), putfloat(3carl).

NAME

btoa - binary to arabic converter for floatsams

SYNOPSIS

btoa [flags] < floatsams > arabic numbers

-RN set sampling rate to N (16384 Hz)
-bN set begin time to N
-eN set end time to N
-f print samples as floating point numbers
-d print samples as decimal numbers
-o print samples as octal numbers
-x print samples as hexadecimal numbers
-t index samples with time in seconds
-s index samples with sample numbers
-F index samples with FFT center frequencies (use only after spect)
-D also print sample values in db
-h usage statement

DESCRIPTION

btoa reads floatsams (32-bit binary floating point samples) on its standard input and prints their arabic equivalents on its standard output. Arabic is printed even if the standard output is a file or pipe.

btoa defaults to writing its output on the standard output, formatting the sample value as a floating point number in the signed unit interval. The **-t** flag causes samples to be indexed by time in seconds at the prevailing sampling rate. The sampling rate defaults to 16384 Hz and is changed by the **-RN** flag where N is the desired sampling rate.

The sample formats are as described in the synopsis. They are cumulative, that is, selecting octal and floating point will print both formats on the same line.

The -F flag should only be used with **spect** in a construction such as

```
sndin file |spect -d |btoa -s -F |more
```

The advantage to this form (as opposed to just letting the output from **spect** appear directly on the terminal) is that it produces only one screenful at a time. A similar usage would be something like

```
sndin file |energy |btoa -s -t -D |more
```

AUTHOR

Gareth Loy

SEE ALSO

atob(1carl), xform(1carl), getfloat(3carl).

NAME

burpsf – free space compaction for csound file system

SYNOPSIS

burpsf filesystem

BRIEF DESCRIPTION

burpsf squeezes the unused bubbles of freespace into one chunk by doing a linear compaction. It first coalesces adjacent free blocks. Then it goes from low addresses on the disk to high, copying files down to fill all the empty spaces.

FULL DESCRIPTION

The routine first reads into core the entire free list and also all the sound descriptor files (SDF). This is because the contiguous files can be spread across the disk, and as a linear compaction will be performed, a sound file descriptor (SFD) may have to be put down at one moment and picked up again later as the disk is bubbled through. It then copies the SDF structures, creating new disk block pointers on a new free list which it makes up. Since the allocation strategy is linear first-fit, it builds the new list from the beginning of the disk. It then determines which files appear in different places on the two lists.

At this point it asks you if you want to see the list of files it wants to move, and the base addresses it wants to move them to. You should routinely look at this list before actually allowing **burpsf** to do it.

After previewing the list, it will ask if you wish to proceed. If the free list looks reasonable, (the block of free space at the end of the new list should equal the sum of fragmented free blocks on the old list) then type

y [return]

and go get a cup of coffee. It can take up to 20 minutes for a 300MB disk if it has to move all the files. **burpsf** shows the progress of the compaction, showing the file, the source block being copied, and the destination. It also gives an indication of the time in seconds it will take to copy the file.

At the end of each block copy, the updated sound descriptor file that points to that block is written out. The current copy of the free list is also written out, with the blocks not yet copied marked as ALLOCATED. Thus, if **burpsf** dies while copying, the free list in /<device>/dskcyls will be erroneous, but the SDF files will be accurate. Since **sfck** can rebuild the free list from the SDF files, this should be a recoverable situation.

Bad Block Files

If you have isolated bad block files as described in the document "Managing Bad Blocks on a Csound File System", **burpsf** will not try to move them. Files will be moved around the bad blocks insofar as they fit.

AUTHOR

Gareth Loy

SEE ALSO

cpsf(1csound), "Managing Bad Blocks on a Csound File System".

DIAGNOSTICS

You must be a member of the csound super user group to run this program. If you are not, it will bluntly inform you of that fact and quit forthwith. It will complain if it fails in coalescing free blocks in advance of figuring out what to copy. This shouldn't happen if you've run **sfck** first and all was well. It will also complain and stop if it encounters anything except USED or UNUSED blocks on the free list, since **sfck** should have caught these as well. If a read or write error is generated copying blocks of a file you will get the error message: "Aaaarrrggh!". **burpsf** then quits. The thing to do at that point is to rebuild the free list. Let **sfck** do it. Move the

file /<device>/dskcyls to some temporary name and run **sfck**, which will notice the absence of dskcyls and regenerate it.

Notice

Running this program must **ALWAYS** follow on a check of the integrity of the disk free list via **sfck(1csound)**. You **MUST** correct any errors that occur there first.

It is not unwise to have done a dump of the disk first, also. Theoretically, the worst that can happen is that a single file is lost if **burpsf** dies in the middle of a copy. Otherwise, the theory is that **sfck** can always put the file system back together. However, with a program that has this much potential for disaster, it is worth taking precautions. Its potential for disaster arises from the fact that it touches every file on the disk.

BUGS

burpsf could be smarter if it first looked for files of the exact size to fill the holes instead of moving all files down one after the other, but it took long *enough* to write this program!

NAME

cannon – continuous distribution function generator

SYNOPSIS

cannon [-LN] [-S[N]] [-I] [-F] [-CN] function arg1 arg2 < floatsams > floatsams

DESCRIPTION

flags: (default)

-L set N output floatsams (1024)

-S set seed for random number generator to N (or to time of day if no N) (no S flag uses default constant seed)

-I use standard input instead of internally generated noise

-F use 1/f noise instead of white noise as input to cannons

-C use correlated noise; N= correlation factor: 0= white, -> 1= brownian

functions: args: (default value)

arcsin	no args
beta	a (0.25) b (0.25)
cauchy	tau (0.01272) iopt (0.0)
exponential	delta (2.0)
gamma	nu (2.0)
gauss	sigma (1.0) xmu (0.0)
hyperbolic	tau (0.48732) xmu (0.0)
linear	g (1.0)
logistic	alpha (2.2021) beta (0.0)
laplace	tau (0.50723) xmu (0.0)
urand	lb (0.0) ub (1.0)
frand	lb (0.0) ub (1.0)
crand	lb (0.0) ub (1.0)
randfl	srate(16384) freq(440)
randfc	srate(16384) freq(440)

cannon produces a stream of random numbers constrained by the chosen continuous distribution function. This program conforms to the **cmusic gen** command syntax, and can be used to fill **cmusic** function tables. This example,

```
% cannon -L1024 gauss > file
```

writes 1024 binary floating point samples into *file* which conform to a gaussian distribution function. A simple histogram of a **cannon** can be produced by using **stochist(1carl)**. For example:

```
% cannon gauss |stochist
```

Here is a cmusic example:

```
var 0 s1 "gauss"; gen 0 cannon f3 s1;
```

The best description of these functions is obtained from [Lorrain], but here is a brief discussion.

The distribution of the output of **cannon** is determined by which distribution function is named. Ordinarily, by naming a distribution function, the output of a noise generator is passed to the named distribution function, which then applies its characteristic distribution pattern to

these numbers.

By default, the *urand* generator is used. The -F flag uses the *frand* generator. There is also a variable-correlation noise generator, *crand*. *crand* is invoked by -CN, where N is a number from 0 to 1 specifying the degree of correlation in the noise. The -I flag causes **cannon** to read *floatsams* from its standard input which are used to drive the distribution functions, so if these three noise generators aren't enough, you can roll your own. (Reading in a ramp from -1 to +1, for instance, produces a neat picture of the actual transform of the cannons). Besides providing random values to the distribution functions, the output of *urand, frand and crand* can be had directly by naming them instead of a distribution function in the proper place on the command line.

> "*Cannon* is a term Xenakis coined to name algorithms 'shooting' random values according to a specific probability distribution. (In French we say *tirer au hasard* for 'choose at random,' literally 'shoot at random.')"
> -[Lorrain]
> Thus, with this program, it is possible to "compose with serialism shot from guns."
> -[Zvonar]

Normalized Symmetrical Distribution Functions

A terse description of the individual distribution function cannons follows. For more details, see [Lorrain].

NOTE: the default arguments for the following four distribution functions have been chosen to adjust them to a common characteristic. F(3), the probability the value of 3, of the gaussian distribution is taken as a standard, and the other three are normalized to have a probability for F(3) that agrees with this. Thus, these four distributions can be used effectively together to produce distributions of the same "size" but of different shape. Although the remaining functions do not submit to the same normalization, normal values for the Cauchy distribution are given by [Lorrain] which are also used by default.

gauss: sigma (1.0), xmu (0.0). Using the defaults, an approximated standard gaussian distribution is produced - the classic "bell curve". Sigma is a scaler controlling dispersion, or range, and xmu is an offset. The mean can be moved away from 0 by varying xmu, and the extension of the distribution function can be affected by sigma. The cannon approximation algorithm is:

```
for (i = 0; i < 12; i++) sum += Us; Xs = sigma * (sum - 6) + xmu;
```

hyperbolic: tau (0.48732), xmu(0.0). Hyperbolic cosine, symmetrical, like gauss, but although it is centered on zero, it has no mean. Thus, it can be viewed as a somewhat civilized Cauchy distribution. The cannon formula is

$$Xs = \tau * \log(\tan(\pi * \frac{Us}{2})).$$

Tau is a scaler, xmu an offset.

logistic: alpha (2.2021), beta (0.0). Another symmetrical distribution of mean – beta/alpha and mode alpha/4. (Mode is the point of highest probability). Alpha and beta thus control the mean and dispersion of the distribution. The cannon formula is

$$Xs = \frac{-beta - \log(1/Us - 1)}{alpha}.$$

laplace: tau (0.50723), xmu (0.0). First Law of Laplace. This is a bilateral exponential distribution (see *exponential,* below for a unilateral distribution). The distribution function has the shape of two exponential slopes back to back intersecting at 0. Tau and xmu are scale and offset. Cannon formula first selects Us in the range [0,2], then for Us > 1.0, it applies $Xs = -tau * \log(2.0 - Us) + xmu$. For Us < = 1.0, it applies $Xs = \tau * \log(Us) + xmu$.

Asymmetrical Distribution Functions

linear: g (1.0). The distribution function goes linearly from 2/g to 0 as x goes from 0 to g. The cannon formula is Xs = g(1 - sqrt(Us). Small values are favored because sqrt(Us) > Us, which tends to make 1 - sqrt(Us) always nearer 0.

exponential: delta (2.0). This is a unilateral exponential distribution. Delta controls the density near 0, with greater values of delta producing greater density near 0, smaller values producing less steep curves. The cannon is

$$Xs = \frac{-\log(Us)}{delta};$$

gamma: nu (2.0). This produces an asymmetrical distribution that is 0 at x=0, reaches its mode at nu - 1, with the mean at nu. The values of nu are integers, greater than 0. Any fractional part is truncated. For nu = 8, it resembles the gaussian function, except that the probability of x at 0 is always 0; this means that for smaller values of nu, the skirt to the left of the mode is very steep, whereas the skirt on the right is roughly exponential. The cannon algorithm is

```
for (i = 0; i < nu; i++) sum *= Us; Xs = - log(sum);
```

The *Poisson* distribution can be approximated by the *gamma* function.

cauchy: tau (0.01272), iopt (0.0). Cauchy is a symmetrical distribution function centered around zero, related to hyperbolic cosine. The Cauchy distribution has the property of having no mean and generating very heterogeneous values, explaining why it is listed under asymmetrical functions. The histogram of this function resembles the gaussian distribution, but the skirts approach the x axis so slowly as to make extremely distant values likely. The cannon formula is

$$Xs = \tau * \tan(\pi * Us).$$

If iopt= 1, the distribution is folded into the positive range only.

beta: a (0.25) b (0.25). This cannon produces various "U-shaped" distributions within the range [0,1]. In general, as a and b go from 1 to 0, the U gets deeper. For a = b = 1, it produces a *continuous uniform* distribution. As a and b go to 0, the probabilities are more and more restricted to the region around 0 and 1. For a = b = .5, we get the *arc sine* distribution (see below). For cases where a != b, the larger value tends to reduce the density in that region. Thus, the values a = .75, b = .5 produces a function that is deeper near 0 than near 1.

arcsin The *arc sine* distribution is the same as the *beta* distribution for values of a= .5, b= .5. The cannon is $Xs = \sqrt{(\sin(\pi * Us/2))}$.

Random Functions

The random number generators used to provide values for the distribution functions can be called directly. They are *urand:* lb (0.0) ub (1.0), *frand:* lb (0.0) ub (1.0), and *crand:* lb (0.0) ub (1.0), for *uniform, 1/f,* and *correlated* noise, respectively. The *urand* generator is simply the UNIX math library rand() function, scaled into the appropriate range as follows:

$$Us = (ub - lb) * (\frac{rand()}{pow(2.0,31.0) - 1.0} + lb.$$

frand, the *1/f* generator, is an implementation of Voss' algorithm as given by [Gardner], which involves summing the output of a collection of rand() generators. New values are then obtained from these generators in a binary-weighted order, that is, one in which successive values are weighted by inverse exponent.

The *crand* generator produces correlated noise by constraining the next value of the generator to lie within a window of the previous value. This window is not to be confused with the upper and lower bounds, which are static boundaries. The size of the dynamic window is scaled by the argument supplied to the -C flag. As the -C flag value, N, goes from 0 to 1, the size of the dynamic window goes from the size of the static window to 0. Thus, as N goes to 1, the range of the next random number lies closer to the previous one, producing the correlated effect. Informal experiments have shown that -C.96 produces roughly the same spectrum as *1/f* noise. The character of the noise is quite different, however.

Other Functions

randfi: srate (16384), freq (440). produces a linearly interpolating random function, at frequency **freq** of sampling rate **srate.** A continuous function is generated by linearly interpolating between successive random values, where the frequency of new random values to interpolate between is set by **freq.** This is the equivalent of the RAN unit generator of MUSIC V.

randfc: srate (16384), freq (440). produces a cosine interpolating random function, at frequency **freq** of sampling rate **srate.** It is like randfi(), but it interpolates with the cosine function between 0 and pi. This produces a smoother effect in the transitions than **randfi()**, and its spectrum shows a smoother high frequency rolloff.

AUTHOR

Gareth Loy

SEE ALSO

[Gardner]: Martin Gardner, "Mathematical Games", Scientific American, March, 1978.
[Lorrain]: Denis Lorrain, "A Panoply of Stochastic "Cannons"", Computer Music Journal, V. 4 #1, p. 53.
[Zvonar]: Richard Zvonar, private communication, 1982.
libran(3carl), stochist(1carl)

DIAGNOSTICS

When using the -I mode of reading in values from the standard input for driving the distribution functions, care must be taken that the source of random numbers is inexhaustible, since some of the cannons use more than one random number to create one output value. **cannon** will say "cannon: ran out of standard input" if this happens, and exit.

NAME

casslab – cassette label maker

SYNOPSIS

casslab [label_file] |**tbl** |**troff**

DESCRIPTION

casslab reads up a *label_file* (or standard input) and outputs lines suitable for ingestion by tbl (whose output should be fed to troff) for producing cassette labels suitable for sliding into the front of a clear cassette box.

The format of the *label_file* is as follows:

```
tape title
<blank line>
side a entries
<blank line>
side b entries
```

Side a/b entries are of the form:

```
tape-counter/time <tab> title
```

Here is a sample *label_file*:

```
Depeche Mode 12" Singles

000     Love in Itself
123     Everything Counts
234     Work Hard!
345     Get the Balance Right!
456     Leave in Silence
567     See You
678     Now, This is Fun

000     My Secret Garden
123     Just Can't Get Enough
234     Any Second Now
345     New Life
456     Shout!
```

Even if you don't have or use the *tape-counter/time* field there must be a tab in front of the title. In the format for the *label_file* the blank lines are required and are not optional even if one side is blank or there is no title for the tape.

Since *casslab* produces input for *tbl* the output of *casslab* can be captured and mangled to one's desires; e.g.

```
% casslab label_file > foo
% vi foo
% tbl foo |troff
```

At CARL the output of *tbl* should be fed to *ltroff* (or *ditroff*) and not *troff*.

BUGS

You have to cut them out yourself.

AUTHOR

Rusty Wright

SEE ALSO

tbl(1), troff(1)

NAME

catsf - concatenate sound files

SYNOPSIS

catsf [-h] [-c] [-rN] source [[-rN] source]... destination

-rN repeat the next file N times

-c force file to be contiguous

-h gives a help message

DESCRIPTION

catsf concatenates the source sound files to the destination sound file. (The sources can be any sound file, including the same one named any number of times.) The -r flag takes an option N of the number of times to repeat the following file.

The attributes of the destination file are determined by the attributes of the first source file. These attributes include packing mode, number of channels and protection. A check is made on all files to be concatenated to make sure they agree in packing mode and number of channels.

The output file also inherits the contiguous/non-contiguous attribute of the first named input file. If it is non-contiguous and you want it contiguous, use the -c flag.

To obtain intervening silences between concatenated files, use files with zeros in them of the appropriate lengths (use e.g., wave(1CARL)).

NOTE: **catsf** always treats the last file in the list as the destination. If you leave this off, you may end up concatenating all the files into the last named source file, thereby loosing it.

AUTHOR

Gareth Loy

SEE ALSO

cpsf(1csound), mvsf(1csound), rmsf(1csound).

DIAGNOSTICS

You may not concatenate a file to itself. You may not concatenate files with conflicting attributes.

BUGS

There is a silent limit of 64 files that can be concatenated at one blow.

If the first file is contiguous the file created by concatenation will be too by default, and if the total size of the file is bigger than the largest block on the disk, you will get the message "allocsf: out of space!".

NAME

catt, cu – connect to remote computer

SYNOPSIS

catt [– **s**speed] [– **l**line] [– **m**machine] [– **w**]
cu [– **s**speed] [– **l**line] [– **w**] telno

DESCRIPTION

Catt and *cu* connect to a remote computer. *catt* is used to connect to a computer on the computer center's dataswitch. *cu* is used to connect to a computer via the automatic calling unit. Both programs manage an interactive conversation with possible transfers of text files.

Both programs program recognize several flags. The – **s** flag may be used to specify the transmission speed (for catt: 110, 150, 300, 1200, etc., for cu: either 300 or 1200). The – **l** flag may be used to specify a name for the communications line (for catt: /dev/dsw0, /dev/dsw1, etc., for cu /dev/acu0, /dev/acu1, etc.). The – **m** flag is only recognized by the *catt* program, it may be used to specify the machine (catta, cattb, eecs, phonlab, vax, 7800). These flags can be used to override the following built-in defaults:

– **s**2400 – **l**/dev/dsw0 – **m**catta (catt)

– **s**1200 – **l**/dev/acu0 (cu)

The – **w** flag is only recognized by the *catt* program, it may be used to specify that the program should terminate if there are no available ports to the requested machine.

Lines beginning with '~' have special meanings and are interpreted as commands. *catt* and *cu* recognise the following commands:

~.	terminate the conversation.
~< file	send the contents of *file* to the remote system, as though typed at the terminal.
~!	invoke an interactive shell on the local system.
~!cmd ...	run the command *cmd* on the local system.
~$cmd ...	run the command *cmd* locally and send its output to the remote system.
~%cd dir	change to the directory *dir* on the local system.
~%speed speed	change the dataswitch line speed to *speed*.
~%take from [to]	copy file "from" (on the remote system) to file "to" on the local system. If "to" is omitted, the "from" name is used both places.
~%put from [to]	copy file "from" (on local system) to file "to" on remote system. If "to" is omitted, the "from" name is used both places.
~:	during an output diversion, this toggles silent mode (initially on) on or off i.e., whether information recieved from the remote system will be written to the standard output. This allows a "progress report" during long transfers.
~?	print a list of the available commands and their syntax.
~~...	send the line '~...'.

catt and *cu* handle output diversions of the following generalised form:

~>[>][:]file
zero or more lines to be written to file on local system
~>

In any case, output is diverted (or appended, if '>>' used) to the file. If ':' is used, the diversion is *silent*, i.e., it is written only to the file. If ':' is omitted, output is written both to the file and to the standard output. The trailing '~>' terminates the diversion.

The use of ~**%speed** requires the existence of the *stty* command on the remote computer.

The use of ~**%put** requires the existence of the *stty* and *cat* commands on the remote computer. It also requires that the current erase and kill characters on the remote computer be identical to the current ones on the local computer. Backslashes are inserted at appropriate places.

The use of ~**%take** requires the existence of the *echo* and *tee* commands on the remote computer. Also, **stty tabs** mode is required on the remote computer if tabs are to be copied without expansion.

FILES

/dev/dsw0
/dev/acu0
/dev/null

SEE ALSO

cu(1), tty(4)

DIAGNOSTICS

Various complaints about local tty line being busy, dead remote system, no available remote system ports, or possible dataswitch error. Exit code is zero for normal exit, nonzero otherwise.

BUGS

Long transfers can sometimes be difficult to terminate prematurely; try using '~:' followed by pressing the DEL or RUBOUT key.

The *catt* program defaults to running at the highest speed possible and transfers may not always work since it may be difficult for the remote system to keep up with such a high transfer rate, particularly if it is heavily loaded. Try reconnecting at 300 baud (*catt – s300*).

NOTE

Since there is an abundance of public terminals for the dataswitch the *catt* program generally should only be used to transfer files, particularly when there is scarcity of available terminals.

NAME

cdsf - change csound file directory

SYNOPSIS

cdsf [flag] [directory]

flag:

-HX cause directory X to be your home directory.

-uX change directory for user X (reserved for superuser)

DESCRIPTION

cdsf provides a mechanism to change *csound* file system directories in a way similar to the UNIX command **cd**. It can also be used to set up a *home* sound file directory.

There is in the *csound* file system a notion of a home directory similar to that in UNIX. For instance: if my login name is dgl, and a *csound* file system exists called */snd*, then my *home csound* directory is */snd/dgl*. In the simple case **cdsf** can change your current *csound* directory. Saying

```
% cdsf /snd/frm
% lsf
```

will cause **lsf** to print out user frm's home directory instead of user dgl's. You may then reference any of frm's files by just its name, without having to include the full file name path. Likewise, the program **pwsf** prints the name of one's current *csound* directory.

By default, your home directory (at CARL) is */snd/your_name*. If your system has more than one *csound* file system, you can make any directory on any filesystem be your home directory with the **-H** flag to **cdsf**. For instance,

```
% cdsf -H/snd1/dgl
```

changes my home directory to default to */snd1/dgl*. (Note there is no space between the **-H** and the directory in this case). Now when I say

```
% cdsf
```

by itself, the directory I will revert to is */snd1/dgl*. To return to your home *csound* directory, either name it explicitly,

```
%cdsf /snd/dgl
```

or say

```
% cdsf
```

by itself.

Subdirectories

The *csound* filesystem supports subdirectories (but see BUGS). You can use **mksfdir(1csound)** to make a subdirectory below your home *csound* directory. You can then change your default directory so subsequent *csound* commands will look there for files by naming the full path to the subdirectory. For example, if I have a subdirectory named *subdir*, I could say:

```
% cdsf /snd/dgl/subdir
```

Other Filesystems

If there is more than one *csound* filesystem available, (at CARL, there are currently two: /snd and /snd1) saying

```
% cdsf /snd1/dgl
```

will change to dgl's directory on /snd1. This could be shortened to

```
% cdsf /snd1
```

to change to your own home directory on /snd1. (The *csound* programs will all fill in the rest of the default path).

File Name Specification

In general, the form of *csound* filenames suitable for **cdsf** consists of three parts:

1) a *device* name, such as /snd or /snd1, followed by

2) an optional *path* component, consisting of a user login name followed by optional sub-directory components, followed by

3) an optional *filename* component.

How csound Parses Filenames

csound goes to great lengths to determine the correct full pathname of the sound file requested. It has two sources of information: *defaults* compiled into the programs and the *filename* that is given to a *csound* command for processing. The *csound* program processing the filename first examines the supplied filename for the three filename components. Any missing components are then filled in from the compiled-in defaults.

The -u Flag

The -u flag can be used by the superuser to initialize the home directory for a user of the sound file system. (This is only needed if it is other than the default compiled-in system name (/snd at CARL).) After running *mksfdir(1csound)* to install a principal directory for the filesystem the user will use, saying

```
% cdsf -Hfilesys -uuser
```

will establish the user's home directory as indicated.

BUGS

cdsf does not recognize "." and "..". This means all directory changes must be to an explicitly named directory.

Because of the implementation, once you use **cdsf** to change a directory, you remain in that directory even if you log out and log back in (unlike UNIX, which returns you to your home directory). To cause you to be returned to your home directory when you log in, put a line of the form:

```
cdsf
```

in your ~/.login file, which will cause you to be returned to your home directory every time you log in.

You won't find out that a directory does not exist until you try to access it.

SEE ALSO

pwsf(1csound), lsf(1csound), mksfdir(1csound).

NAME

cexpr – desk calculator with cmusic-compatible arithmetic functions

SYNOPSIS

cexpr [**-e**] [**expression**] ... [< input] > output

The -e flag suppresses the ?? quotation of undigestible expressions.

DESCRIPTION

cexpr is a desk calculator which uses the same expression evaluator as does *cmusic(1carl)*. It is thus useful for doing interactive calculations analogously to the way those calculations will be interpreted by *cmusic* (but see BUGS below).

cexpr first looks for any expressions on its command line. If there are any, it evaluates them in turn. If there are none, it reads its standard input for expressions. Multiple expressions (blank-separated) are allowed on either the command line or the standard input. The standard input format can consist of (possibly several) lines of (possibly multiple, blank separated) expressions. When reading the standard input, **cexpr** is terminated with [CTRL]-D.

Usage in Text Editors

cexpr can be invoked from your favorite text editor, passing one or more lines with expressions on them to cexpr and capturing the result in their place. For instance, in *vi(1)*, if the cursor is on a line of expressions, the command

:.!cexpr[RETURN]

will send the line to **cexpr**, and replace the line with the results **cexpr** produces.

EXPRESSION TYPES

The following is derived from the current documentation about allowed expressions in *cmusic*. Please refer to current documentation about the capabilities of *cmusic*, since **cexpr** tracks its capabilities.

Expressions may contain several types of operands and operators. Some operands and operators are available only in cmusic (these are indicated in the description below). The current list of expressions is as follows:

NOTE: Expressions must not include blank spaces.

OPERANDS:

numbers

Numbers may have three bases; all are of type float whether they include a decimal point or not.

decimal

Any string of digits, which either includes a decimal point, or which does not include a decimal point.

hexadecimal

If a number does not include a decimal point and starts with the characters 0x, then base 16 is interpreted.

octal If a number does not include a decimal point and starts with the digit 0, base 8 will be interpreted.

OPERATORS:

PARENTHESES

Parentheses must balance, and may be used freely to establish operator precedence. Function arguments should be enclosed in parentheses.

UNOPS

The following are unary operators available in expressions in order precedence, with the first set of operators done before anything in the second set. Unary operations are done

before binary operations, and binary operations are done before post operations.

- Unary minus, as in -3*5

sin,cos,atan,ln,exp,floor,abs,sqrt,rand
These are the standard trigonometric functions sine, cosine, and arctangent, from the UNIX math library, as well as the natural logarithm, exponential, floor, absolute value, and square root functions. Rand is a function which returns a random value between 0 and its (positive) argument.

BINOPS

The precedence is as shown below: ^ and % are done before * and /, and * and / are done before + and -.

^,%
(3^.5 means 3 to the .5 power; 397%17 means 397 modulo 17)

*,/
(5*79.2 means 5 times 79.2; 9/5 means 9 divided by 5 (float result))

+,-
(3+3 means 3 plus 3, 3-8 means 3 minus 8)

POSTOPS

Post operators are done last. They general modify the resulting value of the expression which precedes them.

Hz converts frequency to increment; example: 100Hz = 100 * funclength / srate (uses default funclength and prevailing srate) (cmusic only)

sec converts period (time in sec) to increment; example: 2sec = funclength / (2 * srate) (uses default funclength and prevailing srate) (cmusic only)

S converts samples to seconds at the current sampling rate; example: 100000S = 6.103516 at 16K sampling rate. (cmusic only)

dB converts dB (logarithmic) to linear scale; example: -6dB = 10^(-6/20) = 0.5 (approx.)

K converts K to units; example: 8K = 8 * 1024 = 8192

k converts k to units; example: 8k = 8 * 1000 = 8000

Deg converts degrees to radians; example: 180Deg = (180 / 360) * TWOPI = 3.14159

MM converts metronome marks to seconds per beat; example: 120MM = 60/120 = .5 seconds per beat. (Now available in cexpr as well as cmusic.)

IS computes the sum of the first N inverse terms; i.e., 3invs = 1 + 1/2 + 1/3. 0IS = 0 by definition. (cmusic only)

DIAGNOSTICS

When **cexpr** encounters an error in an expression, it inserts the text of the undecipherable expression in the output stream of evaluated expressions, enclosed in question marks. **cexpr** takes one flag, **-e** which suppresses this ?? quotation.

BUGS

The expression evaluator is not identical to that used in *cmusic*, but is the version from **frmlib(3carl)**.

The field scanning mechanism uses **gets(3)** which doesn't know about comma-separated fields, so these show up as errors.

SEE ALSO

cmusic(1carl), expr(3carl).

AUTHOR

cmusic(1carl) was written by F. R. Moore, **cexpr** was hacked together by Gareth Loy.

NAME

channel - select channel or list of channels from floatsam stream

SYNOPSIS

channel [flags] < floatsams > floatsams or text

flags:

-CN set total number of channels to N

-cS select channels S, which is either:
comma separated list of channel numbers, or
NxM, where N is a channel number, M is a skip count.

-t force text output instead of floatsams

-h terse help message

Channels are numbered from 1.

DESCRIPTION

channel reads floatsams (32-bit binary floating point samples) on its standard input and copies selected channels to its standard output. It assumes a channel-interleaved sample order.

By default, it assumes 1 channel, and selects channel 1, making it have no effect. To extract a selected channel, supply the number of channels as **-CN** where N is the number of channels, and **-cS** where S is a comma-separated list of selected channel numbers. For example,

% wave !channel -C5 -c2,3

will treat the output of **wave** as though it were a stream of 5 interleaved channels, and copy on its output channels 2 and 3.

An alternate channel specification is **NxM** where N is a channel number, and M is a count of the number of channels to skip by, up to the limit of the number of channels set with -C. For example,

% wave !channel -C10 -c1x2

will select channels 1, 3, 5, 7, 9. The notations can be safely combined.

% wave !channel -C10 -c1x5,2 selects channels 1, 2 and 5.

AUTHORS

F.R. Moore, Gareth Loy.

NAME

chmodsf - change csound file protection

SYNOPSIS

chmodsf protection file ...

DESCRIPTION

chmodsf provides a mechanism to change *csound* file ownership in a way similar to the UNIX command **chmod.** The *csound* file protection scheme works like its UNIX counterpart. Protection can also be changed with **visf.**

SEE ALSO

chmod(1).

BUGS

There is currently no way to modify *csound* file directory permissions.

NAME

chownsf - change csound file ownership

SYNOPSIS

chownsf owner file ...

DESCRIPTION

chownsf provides a mechanism to change *csound* file ownership in a way similar to the UNIX command **chown.**

Its use is reserved for the superuser, or the *csound* superuser.

SEE ALSO

chown(1).

NAME

chubby – Chebychev polynomial function generator for cmusic

SYNOPSIS

chubby
-LN
dc_offset amplitude_of_partial_1 amplitude_of_partial_2
amplitude_of_partial_3 ... amplitude_of_partial_N

– LN specifies the length of the output function. e.g., -L1024. Expressions can be used.

DESCRIPTION

chubby is a **gen** function for cmusic(1carl). It is useful in generating functions for waveshaping or nonlinear distortion techniques.

Its first argument specifies the number of the points in the **cmusic** function it is to generate, typically 1024. The rest of the arguments specify the amplitude of each of the partials of the Chebychev polynomial function according to: $Tn+1(x) = 2xTn(x) - Tn-1(x)$ where Tn(0) = 0 Hz component, Tn(1) = fundamental, Tn(2) = first harmonic, etc.

An example shell call:

% chubby -L1024 0 .5 .5

produces a Chebychev polynomial function with equal parts fundamental and first harmonic.

gen 0 chubby f1 0 .5 .5;

produces the same function in the context of a **cmusic** score.

AUTHOR

Gareth Loy, based on an algorithm of Marc LeBrun.

SEE ALSO

cmusic(1carl)

BUGS

All **cmusic** gen functions must accept -c and -o flags for "closed" and "open" mode. **chubby** only usefully produces -o format functions, so these flags are no-ops.

Whether the result of **chubby** is normalized depends on the sum of its arguments being 1.0. It is customary to weight-reduce chubby functions with gen0.

NAME

click - find discontinuities in floatsam streams

SYNOPSIS

click threshold < floatsams > floatsams

DESCRIPTION

click reads floatsams (32-bit binary floating point samples) on its standard input and takes the difference of the current and last sample. If the absolute value of this difference is greater than the threshold level, **click** prints the sample number and the difference on its standard output.

AUTHOR

F. R. Moore

NAME

cmpsig – multiple file/channel comparator with CRT graphics

SYNOPSIS

cmpsig [flags] [files] > text mode graph

or

cmpsig [flags] [ ] < floatsams **>** text mode graph

Input must be a file or pipe. flags:

-t	set text mode of output instead of CRT plot
-cN	set number of channels to N (if reading stdin or using -E)
-lN	set lower bound of display to N (-1)
-uN	set upper bound of display to N (+ 1)
-mN	set upper and lower bounds to + and - N ([-1,+ 1])
-RN	show time instead of sample number using sampling rate N
-aN	display average of N seconds of samples.
-eN	display mean squared energy of N seconds of samples
-sN	skip output by N seconds worth of samples of input
-Cc	show sample value as character c ('A')
-Hc	histogram mode using character c ('-')
-EN	each file has N channels (set -c to sum of channels)

All durations are in seconds. Use postop 'S' for sample times. Arguments may be expressions.

DESCRIPTION

cmpsig is similar to *show(1carl)* save that it will plot more than one channel simultaneously. Each separate channel is plotted with a different character, where the first channel is represented by the character *A*, the next by *B*, etc.

If no files are named, **cmpsig** reads its standard input. In this case, you must explicitly tell **cmpsig** *via* its -c flag how many channels the input floatsam stream contains if it is not mono. (**cmpsig** will ignore its stdin if any files are named.)

If files are named on the command line, these are assumed to contain monophonic floatsam data, and one sample from each is taken in order of their mention on the command line. In this case, channel labeling is done automatically, since the number of channels equals the number of files.

Where it is desired to compare multi-channel files, the -E flag comes in handy. For instance, if file *aa* and *bb* contain stereo floatsams, they can be compared by saying:

```
% cmpsig -c4 -E2 aa bb
```

where -c4 specifies the sum total of channels, and -E2 says that each file contains two channels.

Where two channels fall into the same character position on the graph, the collision is represented by a '*' character. The base character from which the sequence is chosen can be changed with the **-C** flag.

If the CRT graphics start to make you dizzy, -t prints them out in numeric format instead.

AUTHOR

Gareth Loy

NAME

cmt – ucmt – comment / uncomment text

SYNOPSIS

cmt < text > commented_text

ucmt < commented_text > text

CMT

```
/*
 * cmt reads text on its standard input and
 * puts C style comments around it, like this.
 */
```

If the text already has comments, (as for example when the previous paragraph is sent through **cmt**) **cmt** interposes a digit between the slash/star pair, as for instance:

```
/*
 * /1*
 * * cmt reads text on its standard input and
 * * puts C style comments around it, like this.
 * *1/
 */
```

Each application of **cmt** to code thusly commented increments the digit by one, showing that this can be repeated recursively to a considerable depth.

UCMT

ucmt reverses the effect of **cmt**, stripping off the outer layer of comments, and diminishing the digit between any slash/star pairs.

The format of comment that **ucmt** is able to recognize is limited to that generated by **cmt**, and consists of this:

```
< slash> < star> < newline>
< blank> < star> < blank> anything
. . .
< star> < slash> < newline>
```

FAVORITE HACKS

The author's personal motivation for creating these programs is to be able to do the following in *vi(1)*:

```
(discover need to edit verbose comment)
:m,n!ucmt
(modify text of verbose comment)
:m,n!cmt
```

AUTHOR

Gareth Loy

NAME

cmusic – CARL sound synthesis program

SYNOPSIS

cmusic [**-o**] [[**-v**] [**-n**] [**-t**] [**-q**] [**-Rx**] [**-Lx**] [**-Bx**] [score_file] > output

BRIEF DESCRIPTION

cmusic flags given on the command line override options in the score_file:

-o tells cmusic to produce no sample output (debug mode)

-v sets the verbose option; **-v-** turns it off (default = off).

-n sets the notify option; **-n-** turns it off (default = off).

-t sets the timer option; **-t-** turns it off (default = off).

-q turns off any verbose, timer, or notify options set elsewhere.

-Rx sets the sampling rate to **x** (default = 16K).

-Lx sets the default function length to **x** (default default = 1K).

-Bx sets the io block length to **x** (default = 256).

Flag symbols must not be combined, i.e., "-tn" will not work, but "-t -n" will.

If no score_file is given, *cmusic* reads its standard input. If its standard output is connected to a terminal, *cmusic* generates ASCII sample values on the screen; if the standard output is not a terminal, binary values (floats by default) are produced. Detected score errors generally cause sample synthesis to stop – the remainder of the score is scanned for further errors, if possible.

FULL DESCRIPTION

cmusic is a C program which follows the basic form of the MUSIC V sound synthesis program, but with many improvements and extensions. All input to *cmusic* is contained in the score_file, which consists of definitions for instruments and functions, and instructions to play notes on these instruments. If a score_file is given, *cmusic* reads from this file – otherwise the standard input is read. *cmusic* writes samples on its standard output, which can be piped to a subsequent signal processing program, or connected to a file via the UNIX ">" convention or via the "set" command described below. If the standard output is connected to the terminal, ASCII data is produced – otherwise *cmusic* produces binary sample data in either 16-bit integer (DAC-compatible) or 32-bit floating point form (for piping through signal processing programs or to the soundfile programs).

The Score File

The score is essentially a time-ordered list of statements describing actions to be taken by the program. Each statement in the score file begins with an instruction and ends with a semicolon. Instructions are available to define instruments, to generate stored tables (functions), to set certain parameters and options,

MUSIC V conventions are used where possible to permit experienced users to use *cmusic* as quickly and effectively as possible, and because MUSIC V is widely known and documented. Improvements and extensions to MUSIC V include:

Language – The *cmusic* program is written entirely in C, allowing advanced users to supply their own unit generators in an efficient high level language.

Use of memory – *cmusic* is entirely dynamic in its use of memory, using only as much as required by an individual score. *cmusic* is therefore more suitable for a timesharing environment than the batch-oriented MUSIC V.

Reentrancy – Unit generators are reentrant (as in MUSIC V), which allows an

arbitrary number of notes to be played simultaneously on one instrument.

New data types – Instruments and functions are more flexible, and several new unit generator argument types have been added, such as constants, strings, and dynamic variables.

Preprocessor – The C compiler's macro preprocessor is used on the score, allowing the convenience of macros with arguments and "include" statements in the score.

Expressions – Whenever a number is expected, an expression may be given, with numbers, note parameters, or static variables as operands, and with a wide variety of operators, including + , −, *, /, ^ (to the power of), % (modulo), and parentheses. Variables may be given either as simple names such as p8 or v3, or as subscripted named, such as p[8], or v[v[2]], where subscripts may be any constant expression. Mathematical functions include sin(), cos(), atan(), ln(), exp(), floor(), rand() (returns a value between zero and its argument), and abs(). Postoperators include dB (which converts the value which precedes it to linear amplitude), and Hz (which converts whatever precedes it to an oscillator increment corresponding to the given frequency).

"Set" command – New mnemonic commands for setting the sampling rate, number of output channels, etc., are provided.

Variable function length – Variable length functions may be defined and used, and the size of io blocks is under user control.

Summary of Commands

The general form of commands in *Cmusic* is

operation time parameters ;

where operation is one of the basic operations in the list below, time is the time (in seconds) at which this operation is to be done, and parameters depends on the operation chosen. Only the first 3 letters of any operation need be given, although more are permitted (thus the "note" operation may be typed as "not", if desired.

comment – the text following this operation (up to the next semicolon) is taken to be a comment. The text is not interpreted by *Cmusic*.

generate – (see below).

instrument – (see below).

merge – this command allows one or more sections of the score to be sorted and/or merged into time order, as explained below.

note, play – (see below).

print – this command has the general form:

print time parameters ;

and causes the values of the parameters to be printed out on stderr at time. A parameter beginning with a percent sign is interpreted as a string and that string (without the %) is printed out literally.

Example: "print 2.55 %time= p2 ;"

section – (see below).

set – (see below).

terminate – (see below).

variable – This command causes variables (v's or p's) to be set to the specified values at the time specified. For example, "var 3 p[7] 14 15;" sets note parameter p7 to 14 and note parameter p8 to 15 at time 3.

Cmusic ignores any lines in the score which begin with the sharp sign character.

Instrument Definition

Instruments are defined as in MUSIC V, with the general form:

```
ins time name ;
unit generator statements;
end;
```

Unit generator statements have the general form:

UGname parameters ;

where UGname is the name of a unit generator, and parameters defines input and output connections as required by the particular unit generator chosen.
Unit generators are syntactically described by a unit generator name followed by a description of its parameters.
For example, the following description of the "adn" unit generator:

```
adn output[b] input[bvpn]*
```

indicates that its first parameter specifies where its output is to go, and that this parameter must be an io block ("b"). Further parameters to adn are inputs, any of which may be io blocks ("b"), static variables ("v"), note statement parameters ("p"), or constant numbers ("n").
Other possibilities would have included functions ("f"), or dynamic variables ("d").
The asterisk at the end of the parameter description line indicates that an arbitrary number of "input" parameters may be given.

Unit Generators

(N.B. This is not a complete list.)

out input[bvpnd]* The "out" unit generator is the normal destination for the final output signal(s) of an instrument. It must have as many inputs as there are channels of sound output. The signal fed into the first input is summed into channel 1, the second input into channel 2, and so on, up to 4 channels. Inputs may be io blocks, static variables, note parameters, constant numbers, or dynamic variables.

adn output[b] input[bvpn]* The "adn" unit generator sums together an arbitrary number of inputs to produce its output. The allowable types of parameters are as indicated.

mult output[b] input[bvpn]*
The "mult" unit generator multiplies together an arbitrary number of inputs to produce its output. The allowable types of parameters are as indicated.

osc output[b] amplitude[bvpn] increment[bvpn] table[f] sum[dpv]
The "osc" unit generator is *cmusic's* main synthetic signal source. It executes the table lookup oscillator algorithm defined by the relations:

output(I) = amplitude(I) * table[sum(I) % table_length]
sum(I+1) += increment(I)

where output(I) refers to the *Ith* sample of output, table[N] is the *Nth* value stored in the table, table_length is the length of the table (whose indices vary from 0 to table_length − 1), and "%" is the modulus operator.

test condition[vpn] action[vpn] input[bvpdn]*
The "test" unit generator examines its input(s) and takes a specified action if a certain condition is met. The types of input(s) required depend on the particular test being made. A file of macro definitions for conditions and actions is maintained in ~frm/music/test.defs. Conditions include a signal not exceeding a given threshold for a given time, etc. Actions include shutting off a note before its normal termination time in order to economize computation time.

show input[bvpdfn]* This is a special unit generator for debugging instruments. Each time it is executed, the values of its arguments are printed out on stderr. It acts as a very verbose signal probe.

Sample Instrument Definitions

The following statements define one of the simplest possible instruments: a single oscillator. The instrument is named "toot", the oscillator places its output into io block b1, which is connected directly to the "out" unit generator. The amplitude of the oscillator is set by note parameter 5 ("p5"), which is the fifth field in a note statement, directly following the duration. "p6" determines the oscillator increment which controls the frequency according to the relation

frequency = increment * sampling_rate/table_length ,

function "f1" is used as the table, and the sum is stored is dynamic variable d1.

```
instrument 0 toot;
osc b1 p5 p6 f1 d1;
out b1;
end;
```

The following instrument, "bonk", sums the output of two oscillators
places the combined signal in output channel 1.
The second oscillator's output is separately placed into output channel 2.

```
ins 0 bonk;
osc b1 p5 p6 f1 d;
osc b2 p7 p8 f1 d;
```

```
adn b1 b2 b1;
out b1 b2;
end;
```

The dynamic variables need not be indexed.
In the above definition, the first dynamic variable is effectively "d1"
and the second one is "d2".
Indices are needed only if a particular dynamic variable need be referenced
in more than one place within an instrument definition.

Also, the word "instrument" need not be spelled out completely:
in most cases
cmusic
will accept reasonable abbreviations (where reasonable is defined as
"3 or more letters").
Unit generator names must be given exactly as they are stated above,
however.

Function Generation

Functions are generated via the "generate" statement, which has the general form:

generate time gen_option function_name list

where the "gen_option" selects among the various function generating options, "function_name" names the functions to be generated, and "list" depends on the requirements of the generating option. "list" may generally include constant numbers, expressions containing variables or note parameters, or string variables. All MUSIC V generator options are available (1, 2, and 3), as well as some new ones, described below. Function values normally range from −1.0 to + 1.0, and special function lengths can be generated by enclosing the length between square brackets after the function name.

Generate Statement Examples

The statement

```
generate 0 gen2 f1 1 1;
```

specifies that function "f1" shall consist of a single period of a sine wave, and the statement

```
gen 0 gen3 f2 0 1 1 0;
```

specifies that function "f2" shall be a piecewise-linear function with a trapezoidal shape (produced by generator 3).

All gen programs are separate from *cmusic* proper, and may be run as freestanding programs. *cmusic* uses the UNIX "popen" facility to gather the output of gen programs into stored function memory. The user may name any program whatsoever as a generator, as in

```
gen 0 ./mygen f2 arg1 arg2 ... ;
```

This causes *cmusic* to execute the system command

```
./mygen -Lfunc_length arg1 arg2 ...
```

(String variables are evaluated before the system call is made.) The output generated by program "mygen" must be func_length floating-point binary values (the *putfbat(3X)* routine is

especially well-suited to this); these values are read into the space designated to hold the function (f2 in this case).

Note Statements

Notes are played via "note" statements ("play" is a synonym for "note", for the sake of compatibility with northern California). These statements have the general form

note time instrument duration parameters ;

where "time" is the starting time of the note (time values are normally given in seconds), "instrument" specifies the instrument to be used, "duration" specifies the length of the note, and "parameters" depend on the requirements of the instrument. Notes may overlap in time in any desired manner; however, it is an error for the "time" of a note statement to be less than that of a preceding note statement within the same section of a score. By MUSIC V convention, all fields in the note statement have parameter numbers, with the "time" being the second parameter ("p2"), the duration being "p4", and the first parameter following the duration being "p5". Parameters retain the last value to which they were set in subsequent statements. Thus

```
note 1 ins 3 . . .
note p2+ p4 ins 5 . . .
```

causes the second note to start at time 4.

The Merge Command

A score may contain statements with the general form:

```
merge;
      (any mixture of "note",
      "variable", and "generate" statements,   MERGE SECTION 1
      not necessarily in time order);
  endsec; {end of merge section 1}
      (any mixture of "note",
      "variable", and "generate" statements,   MERGE SECTION 2
      not necessarily in time order);
  endsec; {end of merge section 2}
   .
   .                          ANY NUMBER OF MERGE SECTIONS
   .
endmerge; {end of entire merge}
```

Every statement in each merge section is written onto a temporary file after its action time (p2) has been evaluated (p2 might be given by an expression in the score).
In the case of "note" or "play" statements, the duration (p4) field is also evaluated before the statement is written onto the temporary file.
The temporary files are then separately sorted and merged together into a single time-ordered list.
The commands "instrument", "merge", "section", "set", and "terminate" may NOT appear inside any merge section.

Set Statements

A variety of set statements can be used to "set" various *cmusic* parameters and options. Most such statements are obvious in their effect. Set statements have either the form

set parameter = value;

or the form

set option;

Available set statements include:

set blocklength = N;
sets the length of the io blocks to the constant N (the default block length is 256),

set floatout;
set nofloatout; directs *cmusic* to produce 32-bit floating point or 16-bit fixed-point output samples (floatout is the default mode),

set funclength = N; sets the default length of all functions (the default function length is 1024),

set listing = file;
set nolist; causes an input listing to be generated and placed the specified file, or else turns off such a listing (nolist is the default mode),

set barefile = file; causes cmusic to generate a "bare" form of the score on the named file which has all macros and expressions interpreted and evaluated,

set nchan = N;
set stereo;
set quad; all set the number of output channels (the default number of channels is one),

set output = file; directs the output samples to the specified file (the output samples are normally placed on the standard output),

set plot;
set noplot; starts or stops crude terminal screen plotting, (noplot is the default mode),

set srate = N;
set sampling_rate = N;
set rate = N; all set the sampling rate to the specified value, (the default sampling rate is 16*1024 samples per second),

set verbose;
set noverbose; starts or stops progress reports to the terminal (noverbose is the default mode).

set notify;
set nonotify; send the termination message (only) to the terminal when cmusic is finished. (nonotify is the default mode).

set timer; causes timer messages (seconds of score completed preceded by colons) to be listed on the user's terminal.

set sfbufsize = N;
sets the length of the buffer used by the sndfile unit generator (which reads in sound files) to N BYTES. The default value for this buffer size is 16K bytes (equivalent to 8K 16-bit samples, or 4K float samples).

set tempowith vN;

set offsetwith vN;
causes cmusic to use the specified variable to set the tempo or time offset. If tempo is specified with, say, v1, then the values of both p2 and p4 will be REPLACED with p2*v1 and p4*v1, respectively. If time offset is specified with, say, v2, then v2 will be added to all p2 values after they have been scaled by tempo, if any.
Sample Usage:

```
#define TEMPO set tempowith v10; var p2 v10
#define T (p2+p4)/(v10)
TEMPO 30MM;
note 0 one 1 440Hz 0dB;
note T one 1 440Hz 0dB;
note T one 1 440Hz 0dB;
```

In this example, variable v10 is used to set the tempo of the score.
It is set up and defined with macro TEMPO.
Note the use of v10 in the T macro.

Sample Scores

Score files are piped through the C compiler's preprocessor, allowing "#include" and "#define" statements to be used freely. Comments may be used either as semicolon-terminated "comment" statements, or between (nestable) curly braces ("{}"). Sections may be ended with a "section time" statement which resets time to zero (this allows sections of a score to be easily reordered). If no "time" is given, the section will automatically finish when the last note has stopped playing. The entire score must be terminated with a "terminate time" statement, again with automatic time provided if not stated explicitly. "Section" and "terminate" statements may be used to generate silence at the beginnings and ends of sections or scores.

The following is a sample test score for *cmusic:*

```
set list = m.list;
#define NOTE(time,dur) note time bonk dur
#define Pitch(step,octave,ref) \
ref*2^(step/12)*2^(octave)Hz
#define A(octave) Pitch(0,octave,440)
#define As(octave) Pitch(1,octave,440)
#define Bb(octave) As(octave)
set blocklength = 32, funclength = 32, srate = 32;
set stereo;
#include "ins.sc"
ins 0 bonk;
osc b1 p5 p6 f1 d;
osc b2 p7 p8 f1 d;
adn b1 b2 b1;
out b1 b2;
end;
gen 0 gen2 f1 1 1;
```

```
sec .5;
{
note 0 bonk 10 .25 1.2 .1 2;
NOTE(0, 10) .25 1.2 .1 2;
}
note 0 bonk 5 -12dB (3/2)Hz .1 2;
note 2.5 bonk 5 .25 1.7 .1 4;
note 5 bonk 5 .25 1.9 .1 3;
sec; ter 1;
```

The following test score illustrates the use of the "test" unit generator to turn off a note early:

```
#define DEADSIG     1
#define TERM 1

#define DEAD(sig,thresh,time) \
test DEADSIG TERM sig thresh time d

set blocklength = 16, funclength = 64, srate = 32;
ins 0 honk;
osc b2 p5 p6 f2 d;
osc b1 b2 p7 f1 d;
out b1;
DEAD(b1,.5,.1);
end;

gen 0 gen2 f1 1 1;
gen 0 gen3 f2 1 0;

note 0 honk 2 1 .5 4;
ter;
```

FILES

Some naming conventions are helpful, but not enforced by cmusic:
Scores may be kept on files with names of form "*.sc".
Listing files are often named "*.list".

AUTHOR

F. R. Moore

SEE ALSO

The Technology of Computer Music, by Max V. Mathews, with the collaboration of J. E. Miller, F. R. Moore, J. R. Pierce, and J.-C. Risset, MIT Press, 1969.

The CARL Startup Kit, by the CARL staff, which contains some tutorial introductions to cmusic, the CARL soundfile system, signal processing, etc.

There are also several "help" files which list available unit generators, gens, etc.

frm(3carl), getfloat(3carl), hist(1carl), newwire(1carl), etc.

BUGS

No bugs are permitted in this program.

NAME

convert - time-varying and arbitrary sample-rate conversion

SYNOPSIS

convert -rN [-flags] [filename] < floatsams > floatsams

flags:
r = output sample rate (must be specified)
t = minimum time increment (time-varying only)
Q = quality factor (1, 2, 3, or 4: default = 2)
R = input sample rate (automatically read from stdin)
b = starting sample (0)
e = final sample (end of input)

DESCRIPTION

This program out-performs the standard *srconv* program in three different ways:

1) For simple integer-ratio sample-rate conversion, it produces a result equivalent to the %% specification of *srconv* with less computation.

2) It performs arbitrary sample-rate conversions which are impossible with *srconv*.

3) It performs time-varying sample rate conversions which are impossible with *srconv*.

For non-time-varying applications, *convert* requires only that the desired output sample rate be specified via the -r flag. For example,

sndin file1 |convert -r24K |sndout file2

will produce a file2 which is a 24KHz sample-rate version of file1, regardless of what the sample rate of file1 may be. The default quality factor should be good for virtually any application, and persons using a higher setting should be prepared to prove their ability to hear the difference in a blind listening test!

For time-varying sample-rate conversion, the situation is more complicated. The time-varying specifications are input via a unix file (specified on the command line after whatever flags may be present) of x,y pairs where x is the time in seconds. The y values may either be desired_output_sample_rate_at_time_x, or (input_sample_rate / desired_output_sample_rate_at_time_x).

If the y values are of the first form (i.e., output_rate), then the -r flag has no real meaning. However, its presence is still mandatory. In this case, the -r flag MUST specify the minimum output sample rate (i.e., the minimum y value in the file).

If the y values are of the second form (i.e., input_rate / output_rate) then the -r flag MUST be replaced by the -t flag. This is the only case in which the -r flag is not used and the only case in which the -t flag is used. Furthermore, the -t flag MUST specify the minimum y value in the file.

The advantage to this second form is that when using *convert* after the phase vocoder to change a time-varying time-scaling into a time-varying pitch-transposition, the same x,y pairs can be used to control both *pvoc* and *convert*. For example, to produce an upward glissando of one octave over four seconds:

gen4 -L100 0 1 0 4 2 |btoa -t -R25 |atob > gliss

sndin file |pvoc -T1 gliss |sndout temp

sndin temp |convert -t1 gliss |sndout file

When using *convert* after the phase vocoder to change a fixed time-scaling into a fixed pitch-transposition, a little bit of subterfuge is required. For example, to produce a pitch

transposition of 1.01 the following steps would be necessary:

```
sndin file |pvoc -N1024 -I101 -D100 |sndout temp

sndin temp |convert -R16547.8 -r16K |sndout file
```

The first step produces a time expansion of exactly 1.01 (I/D where I and D are integers chosen to be as large as possible but still less than N/8). If the input file had a sample rate of 16K, then the file "temp" also has a sample rate of 16K. However, if the file "temp" is played at a sample rate of 16547.8 (1.01*16K), then it will be transposed in pitch by a factor 1.01 and shortened back to the duration of the input file. But we want "temp" to sound transposed when played at a sample rate of 16K. Hence, the trick is to sample-rate-convert "temp" from 16547.8 to 16K. Since *convert* normally reads the input sample rate from the header of the input soundfile, the -R flag is used to override the information in the header.

DIAGNOSTICS

AUTHOR

Mark Dolson

FILES

BUGS

NAME

convolve – program for performing FIR filtering by fast convolution

SYNOPSIS

convolve filter_file < floatsams > floatsams

DESCRIPTION

This program performs filtering by the overlap-add fast convolution algorithm. This program is for FIR filters only! Its virtue is that it provides a considerable savings in computation, but only when the filter impulse response is greater than, say, 20. For FIR filters shorter than this, and for all IIR filters (e.g., filters produced by *lpc*), *filter* should be used instead.

convolve expects the command line to contain the name of a standard filter file (such as produced by *fir*). The input and output data streams must be floatsams (binary, floating-point sample values). The filter delay is compensated for so that input and output are precisely aligned in time.

AUTHOR

Mark Dolson

SEE ALSO

fir(1carl), lpc(1carl)

BUGS

NAME

convolvesf – convolve stdin with a soundfile containing a room response

SYNOPSIS

convolvesf [flags] impulse_response_ soundfile < floatsams > floatsams

flags:

g = gain factor (1.)
b = begin time in impulse response (first sample to use) (0.)
e = end time in impulse response (last sample to use) (end)
d = duration of impulse response (end - begin)

DESCRIPTION

convolvesf is identical to *convolve* except that where *convolve* expects a filterfile *convolvesf* expects a soundfile. (Also, *convolve* eliminates the filter delay whereas *convolvesf* does not, but this is very minor.) If the soundfile contains the impulse response of a room or of some resonator, then the effect is to provide the input signal with precisely the reverberation which would have resulted had the sound really been produced in that room or resonator. If only a part of the soundfile contains the desired impulse response, this part can be specified with -b and -e flags. It may also be necessary to adjust the overall gain (-g) by trial and error. The program tries to do something sensible about this, but it does not always succeed.

The one problem with this program is that it takes an incredible amount of memory, and impulse responses longer than 1.3 seconds at 48KHz may have trouble running. For a 16KHz sample rate, this translates to a 3.9 second maximum; but longer responses may work in either case. For information about generating impulse responses which are useful as synthetic room responses, see the helpfile for this program.

AUTHOR

Mark Dolson

When the **ignore** argument to **crack** is 0, flags scanned that are not in the **flags** variable generate an error message and **crack** returns EOF. If **ignore** is nonzero, unknown flags do not generate error messages. The purpose of ignoring flags is so that more than one part of a program can scan the command line without having to know about the flags of all the other parts. For instance, where a program calculates a sampling rate by one flag and a value in seconds in another, it must search for the sampling rate first, then the time value. Two sets of calls to **crack**() would be required, one to scan just for the flag setting sampling rate (since it could occur anywhere among the flags, all flags must be read), another to ignore the rate flag, but to set the time value based on the sampling rate.

NOTE: When making muitiple scans of the command line in a program, it is necessary to reset **arg_index** to 0 to enable **crack**() to rescan all arguments again.

FILES

/usr/local/lib/libdgl.a, /usr/local/lib/libcarl.a, /usr/local/lib/libsf.a.

AUTHOR

Gareth Loy

SEE ALSO

commandline(1carl)

BUGS

When ignoring unknown flags, if an unknown flag has an option associated with it, the option is also ignored. Care should be exercised here because it may be possible that the associated "option" is really more concatenated flags. These, if any, are lost. The rule is that, when ignoring unknown flags, the first instance of an unknown flag causes that flag and the rest of that argument to be discarded. For instance, if flags is set to "abtd", and a command line: "-zcda" is supplied, c d and a would be ignored because they come after z in the same argument. The point is there is no way to disambiguate flags from unknown options when ignoring flags, so concatenating options, while nice, is problematical.

FILES

Demonstration program is: /mnt/carl/src/demo/commandline.c at CARL.

NAME

csound - sound file management system.

SYNOPSIS

burpsf - free space compaction for csound file system
catsf - concatenate sound files
cdsf - change csound file directory
chmodsf - change mode of a sound file
chownsf - change owner of a sound file
closesf - command level program to close a sound file
cpsf - copy a sound file
dumpsf - dump sound files to tape
grepsf - look for sample pattern in a sound file
holdsf - set status of sound file to HOLD
keepsf - set status of sound file to KEEP
locksf, unlocksf - lock/unlock a sound file system
lsf - list sound files, sound file directories
mixsnd - mix sound files
mksfdir - make a sound file directory
mountsf - mount a csound volume
mvsf - move sound file
newsfsys - make new sound file system
opensf - closesf - user-level commands to open/close sound files.
play - play sound file(s) through DACs
purgesf - list files that can be reaped
pwsf - print working sound file directory
reapsf - reap (remove) old, unreferenced sound files
record - record sound file through ADCs
restorsf - restore sound files from magtape
retrosf - output the retrograde of a sound file
rmsf - remove sound file(s)
scratchsf - set status of sound file to SCRATCH
sdc - print usage statistics of a sound file system
sfck - check sound file system for soundness
sfdt - print csound dump statistics
sfnorm - write normalized samples on the standard output
sndcmp - compare two sound files.
sndhist - produce histogram of sound file
sndin - read csound files onto standard output
sndout - write sound files.
umountsf - unmount a csound volume
visf - edit sound file parameters

DESCRIPTION

The **csound** file system was developed to address three distinct weaknesses which UNIX has in recording, storing, manipulating and playing back high quality digital sound. The problems are data volume, data bandwidth and bookkeeping. Each of these problems goes beyond what can be handled conveniently by the regular file system, so a special file system just for sound has been constructed, called the **csound** file system.

The commands which make up the system are described above in the Synopsis. Manual pages exist for them all. There is a tutorial as well. A common library for all these programs enforces a common management discipline on a raw partition of a UNIX file system which is used for sample storage. Regular UNIX files in a small partition on the same physical medium contain

text descriptions of the sample data.

AUTHOR

Gareth Loy

SEE ALSO

"Managing Bad Blocks On A Csound File System", "CARL Real Time Sound File System", "Managing A Csound Tape Dump Regimen", "Introduction to the Csound File System", "Csound File System User Program Example", Gareth Loy.

BUGS

The **csound** file system emulates the structure and behavior of the UNIX file system to a certain degree. Differences are necessary both to disambiguate the two file systems, and also because of the different requirements of the data. Some differences are due to the nature of the environment of the shell: the **csound** file system has no access to such things as "*" and other regular expression syntax. The handling of directories is not a full emulation of UNIX. For instance, there is no notion of ".." although "." is supported by most programs. The weak implementation of directories in past versions has been improved substantially.

NAME

cspline – smooth curve interpolator for cmusic

SYNOPSIS

cspline len_flag x0 y0 x1 y1... xN yN

where len_flag is e.g.: -L1024.

DESCRIPTION

cspline is an implementation of a "gen" function call for *cmusic(carl)*. **cspline** is similar in behavior to *spline(1g)* except that it takes pairs of numbers from the command line (rather than the standard input) as abcissas and ordinates of a function. It produces a similar set, which is approximately equally spaced and includes the input set, on the standard output. The output format is binary floating point, suitable for input to cmusic. If the standard output is a terminal, the decimal values of the function are printed instead. The cubic cspline output (R. W. Hamming, *Numerical Methods for Scientists and Engineers*, 2nd ed., 349ff) has two continuous derivatives, and sufficiently many points to look smooth when plotted, for example by *graph*(1). It is similar to *cmusic's gen4* smooth function generator, except that whereas for *gen4* you must supply transition parameters to create a smooth function, **cspline** accomplishes this for you automatically.

Here is a sample call from a shell:

% cspline -L1024 0 0 .1 .1 .2 1 .3 0

and a sample statement in a cmusic score:

gen 0 cspline f1 0 0 .1 .1 .2 1 .3 0;

In addition, the same options are available as for **spline**, but the syntax is different:

cspline [flags] x0 y0 x1 y1... xN yN

where a flag is -xO, x is a character, O is an argument to the flag, concatenated to it, e.g.: -l1.4.

-k The constant *k* used in the boundary value computation

By default *k* = 0.

-p or -c

Make output periodic ("cyclic"), i.e. match derivatives at ends. First and last input values should normally agree. Note: for cspline this usually doesn't work because of the rescaling done by the linear interpolation stage.

Other Considerations

Whereas for *spline* the number of output points is only approximate, for **cspline** the number of output points is constrained to be *exactly* the number specified. In those cases where the spline algorithm supplies fewer or more points than specified, this is accomplished by linear interpolation to stretch or shrink the function to fit.

Whereas *spline* produces as its output function the *retrograde* of its input function, **cspline** produces the function in its prime form, which is how you want it for cmusic and most other applications.

Given the differences, and the automatic nature of the smoothing process, it is wise to preview the function generated by **cspline** with *graph(1)* or *show(carl)* before using it in a score.

SEE ALSO

cmusic(carl)

DIAGNOSTICS

When data is not strictly monotone in *x*, **cspline** throws up, (unlike *spline*, which simply returns the function you sent it).

BUGS

A limit of 1024 input points is enforced silently (upped from 1000 points silently enforced in *spline)*.

The result of the spline function may not entirely lie in the signed unit interval [-1,+ 1]. Stuff it through gen0 to normalize it.

BUGS

cmusic gen functions must accept -c and -o flags for "closed" and "open" mode. **cspline** only usefully produces -o format functions. If you supply a -c flag, chances are the function it produces will be slightly more periodic, but because the functions are modified to fit exactly the number of output points requested, the periodicity will usually be lost.

NAME

denoise – experimental noise reduction program

SYNOPSIS

denoise [flags] noise_soundfile < floatsams > floatsams

flags:
R = input sample rate (automatically read from stdin)
N = # of bandpass filters (1024)
M = analysis window length (N-1)
L = synthesis window length (M)
D = decimation factor (M/4)
b = begin time in noise reference soundfile (0)
e = end time in noise reference soundfile (end)
d = duration in noise reference soundfile (end - begin)
t = threshold above noise reference in db (0.)
s = amount of temporal smoothing to use: 1 to 8 (1)
r = reduction in dB from average in smoothing (20)
w = weight factor for successive weights in smoothing (.5)
soundfilename must follow all flags

DESCRIPTION

This program is still being fine tuned, so please don't bring in all your old tapes just yet! The only reason that it is being publicized at all is so that people with an urgent need and with low fidelity requirements can use it. Basically, this program tries to reduce unwanted background noise by setting up a bandpass filter bank, and placing a noise gate in each filter. The noise gates shut down independently whenever the energy level in that filter falls below a predetermined noise floor. A soundfile with at least .25 seconds of only background noise must be listed in the command line in order for this noise floor to be computed.

At present, there are only two flags which are at all useful. The The first is the -t flag which specifies the number of dB above the noise floor at which the threshold for noise gating is to be set. This flag is critical. A value of 25 may be an appropriate first guess, but it may need to be raised or lowered by trial and error. The second flag is the -s flag which specifies the amount of temporal smoothing which is to be introduced (this eliminates some of the annoying sonic artifacts of the gating). A value of 1 may be fine for a first try; but a value of 4 may be better if the first try has lots of whistling or burbling. In general, it is better to try and find the best threshold with -s1, and then try -s4.

AUTHOR

Mark Dolson

NAME

derivative – produce (approximation to) derivative of floatsam function

SYNOPSIS

derivative < floatsams > text or floatsams

DESCRIPTION

derivative reads floatsams from its standard input and produces the signal

output[n] = input[n] - input[n-1]

on its standard output in text format (if talking to a terminal) or as more floatsams (if looking at a file or pipe). The output is a good approximation to the derivative of the input except at high frequencies.

AUTHOR

Gareth Loy

BUGS

It necessarily writes one less sample than it reads.

NAME

dumpsf - dump sound files to tape

SYNOPSIS

dumpsf [flags] < files

Flags:

-v verbose, tells what files it is opening,
and on what tape they will go
-f output file, if other than /dev/nrmt12;
-d density to use in calculating the length of the dump
if other than 1600 bpi.
-i source of filenames instead of standard input;
-s tape size, in feet;
-n notify operators when dump needs servicing.
-0-9 dump level, default: 9;
-u suppress update of sfdumpinfo file
-l lock the sound file system while the dump is in progress.
-e estimate the amount of tape the dump will require, no dump.

DESCRIPTION

dumpsf reads a list of sound file names off the standard input to be dumped to tape. Filenames must be separated by newlines. The tape must be mounted before running the program. After checking the files for readability and protection, they are dumped. If the dump requires more than one tape, you are told how many will be needed. You are also given the date, level of dump and a unique tape id number to write on each tape of the dump. The tape id is unique within each filesystem.

If the dump requires more than one tape, **dumpsf** rewinds the tape, and tells you to mount the next tape. It then waits until you have done so, asking "Is the next tape mounted and ready to go?" Type "y" and press [RETURN] when ready and it will continue.

The format that **dumpsf** writes is as follows: On the first tape, a text file containing a directory of all files on the dump is made, followed by EOF. The directory is preceded by a text header indicating the nature of the dump. After this, and on each subsequent tape in the dump, a directory of all files on this tape is made, followed by EOF. From there to the end of the tape (signified by two EOFs), the sound files are written. In writing a sound file, first the sound descriptor file is written, then EOF, then the binary samples and another EOF.

FILES

One file in each mounted filesystem named /<device>/sfdumpinfo contains the records of when various level dumps have been done.

AUTHOR

Gareth Loy

SEE ALSO

restorsf(1csound), sfdt(1csound). "Managing a Csound Dump Regimen".

DIAGNOSTICS

A notice of the dumpdate, level and unique tape id is given. Files that cannot be dumped are noted. Errors in writing the magtape or reading the sound file system are noted and halt execution. It prints "The dump is done" when finished.

BUGS

There may be problems related to restoring a list of files that spread across more than one tape at a time. The symptom is a read error when the second tape is mounted. To get around this, run **restorsf** for each batch of files on each separate tape. **restorsf** knows how to start up on other than the first tape of a dump series.

NAME

energy – plot mean square energy level of function

SYNOPSIS

energy [-wN] [-sN] [-RN] < floatsams > output

flags: (default)

-wN set window size to N seconds (default: .01)

-sN skip output by N seconds (default: half of window duration)

-RN set sample rate to N (read from stdin)

Arguments may be expressions. Use postop 'S' for samples. If output is a file/pipe, floatsams are written. If output is a terminal, values are printed on screen.

DESCRIPTION

energy produces the root mean square of the input over a specified window duration. By default, the window is shifted by half its duration after every output. The **-w** *(window)* flag causes the mean to be calculated over a window of size N, specified in seconds (use the postoperator S for time in samples). The larger the window, the smoother will be the output function. The optimal size of the window is between one and two waveform periods of the input signal. (NOTE: The window is always centered about the "current" sample. For example, with -s100 the window is first centered at 0S, then at 100S, 200S, etc.)

The **-s** *(skip)* flag effectively controls the output data rate. The mean square average is still computed for each sample, but it is reported less frequently. This decimates the output function, reducing the number of output samples by a factor of 1/N. The **-R** flag changes the prevailing sampling rate.

NOTE: the window size and skip size are set in units of *seconds* at the prevailing sampling rate. Postoperators may be used to cause these times to be interpreted as sample values. For example, -s512S will make the skip factor be 512 samples.

AUTHOR

Mark Dolson

NAME

envanal – data reduction by hierarchical syntactic function analysis.

SYNOPSIS

envanal [flags] < floatsams > output

DESCRIPTION

Flags: (default values in parenthesis)

-dN set time-domain length greater than or equal to which an input segment causes a break (.01).

-yN set amplitude difference greater than which an input segment will cause a break (.01).

-sN set maximum size to which a segment can grow before causing a break (.1).

-DN pass 2 version of -d (.1).

-YN pass 2 version of -y (.2).

-SN pass 2 version of -s (.4).

-nN max. # of segments expected (8K segments).

-1 output pass one analysis only.

-2 output both passes (default).

-v verbose mode.

-g produce gen1 mode output, text [x,y] pairs.

-f produce floatsam output (output must be file or pipe).

-pN produce crude CRT plot of output, skipping by N (1 sample).

-PN enable peak detector with threshold of N (.01).

-RN set sampling rate to N.

-h print help message.

This program is based on Strawn's algorithm [1] of syntactic analysis.

It implements a two level grammar for a data reduction by the recursive recognition of line segment features of an input waveform. It produces on the standard output either a summary of these features, a plot of the resulting data-reduced function, or the floating point binary representation of that function. In addition, a peak-reading algorithm has been added that causes the analysis to proceed along the tops of the input segments.

AUTHOR

Gareth Loy

SEE ALSO

[1] John Strawn, "Approximation and Syntactic Analysis of Amplitude and Frequency Functions for Digital Sound", CMJ, V4, #3.

BUGS

The number of expected segments must be supplied. If the input data exceeds this, the program bombs. Since this number is one of the things the analysis hopes to yield, this is a tautology. Meanwhile, one is stuck with guessing.

NAME

envelope – plot high-resolution temporal envelope of a signal

SYNOPSIS

envelope < floatsams > floatsams

DESCRIPTION

envelope produces on stdout a very high-resolution temporal envelope of the signal on stdin. Mathematically, the envelope is obtained as the magnitude of the analytic signal

$$z(n) = x(n) + j x^*(n)$$

where $x(n)$ is the input signal and $x^*(n)$ is the hilbert transform of the input signal. This envelope is like the envelope produced by *energy*, but it has none of the temporal smearing which typifies envelopes produced by *energy*. If such smearing is desired, it can be introduced in a controlled fashion by filtering the output of *envelope* with an appropriate filter. In the absence of filtering, the output of *envelope* is of such high resolution that it can be played and recognized. Whether this has any musical application is questionable.

AUTHOR

Mark Dolson

NAME

equtemp – list or match frequencies to equal tempered pitches

SYNOPSIS

equtemp [frequency]*

DESCRIPTION

equtemp with no arguments prints the Hz values of 88 equal tempered pitches from A0 to C8. It lists the pitch index number (0 to 88), the pitch name, {A, Bb, B, C, CS, D, Eb, E, F, FS, G, GS}, the frequency of the pitch in Hz.

If arguments appear, they must be frequency values (Hz assumed). **equtemp** finds the closest equal tempered pitch to each frequency in turn. If the matched frequency does not land exactly on an equal tempered pitch, then the cents sharp or flat is also given.

AUTHOR

Gareth Loy

NAME

expr - cmusic-like expression evaluator

SYNOPSIS

```
float expr(expression)
char *expression;
extern int exprerr;
```

DESCRIPTION

expr() provides expression evaluation similar to that in *cmusic(1carl)*. It takes the string of the expression to be evaluated and returns the floating point result. Expressions may contain several types of operands and operators. Expressions must not include blank spaces. The syntax of expressions is a subset of that available in *cmusic*. The current list of possibilities is as follows:

OPERANDS:

NUMBERS Numbers may have three bases; all are of type float whether they include a decimal point or not.

decimal
: Any string of digits, which either includes a decimal point, or which does not include a decimal point.

hexadecimal
: If a number does not include a decimal point and starts with the characters 0x, then base 16 is interpreted.

octal
: If a number does not include a decimal point and starts with the digit 0, base 8 will be interpreted.

OPERATORS:

PARENTHESES
: Parentheses must balance, and may be used freely to establish operator precedence. Function arguments should be enclosed in parentheses.

UNOPS
: The following are unary operators available in expressions in order of precedence, with the first set of operators done before anything in the second set. Unary operations are done before binary operations, and binary operations are done before post operations.

{sin,cos,atan,ln,exp,floor,abs,sqrt,rand}
: (These are the standard trigonometric functions sine, cosine, and arctangent, from the UNIX math library, as well as the natural logarithm, exponential, floor, absolute value, and square root functions. Rand is a function which returns a random value between 0 and its (positive) argument.)

{-} (Unary minus, as in -3*p5)

BINOPS

The precedence is as shown below: ^ and % are done before * and /, and * and / are done before + and -.

{^,%} (3^.5 means 3 to the .5 power; 397%17 means 397 modulo 17)

{*,/} (5*79.2 means 5 times 79.2; 9/5 means 9 divided by 5 (float result))

{+,-} (3+3 means 3 plus 3; 3-8 means 3 minus 8)

POSTOPS

Post operators are done last. They generally modify the resulting value of the expression which precedes them.

dB	converts dB (logarithmic) to linear scale. Example: -6dB = 10^(-6/20) = 0.5 (approx.).
K	converts K to units. Example: 8K = 8 * 1024 = 8192.
k	converts k to units. Example: 8k = 8 * 1000 = 8000.
Deg	converts degrees to radians. Example: 180Deg = (180 / 360) * TWOPI = 3.14159.
MM	converts metronome marks to seconds per beat; example: 120MM = 60/120 = .5 seconds per beat.
IS	computes the sum of the first N inverse terms, i.e., 3invs = 1 + 1/2 + 1/3. 0IS = 0 by definition.

FILES

/usr/local/lib/libcarl.a.

AUTHOR

F. R. Moore

SEE ALSO

sfexpr(3carl), cexpr(1carl), xform(1carl).

BUGS

The expression format is a subset of that in *cmusic(3carl)*. Not all things available there are necessarily available here. In particular, postoperators that depend on sampling rate are not available, nor are P fields, V variables, functions, blocks, etc.

NAME

fastfir – program for easy design of simple FIR filters

SYNOPSIS

fastfir

DESCRIPTION

fastfir [**flags**] filter_file [> floatsams]

flags: (defaults in parenthesis)

-n	length of impulse response in samples (127)	
-x	filter type: (1)	
	1	lowpass
	2	highpass
	3	bandpass
	4	bandstop
-w	window type: (6)	
	1	rectangular
	2	triangular
	3	Hamming
	4	generalized Hamming
	5	Hanning
	6	Kaiser
	7	Chebyshev
-f	filter cutoff frequency (use two -f's for bandpass or bandstop)	
-a	Kaiser only: stopband attenuation in db (70)	
-b	generalized Hamming only: w(i) = b + (1-b) * cos(2*pi*i/(n-1))	
-c	Chebyshev only: desired filter ripple attenuation in db (70)	
-d	Chebyshev only: transition width (0)	
	(for Chebyshev, any two of -n, -c, and -d is sufficient)	

if filter_file is not specified the file tmp.flt will be created

This program designs lowpass, highpass, bandpass, and bandstop FIR filters by the classical windowing technique. It is much easier to use than *fir* and produces filters which are nearly as good. The impulse response is written into the specified filter_file in standard format. The resulting filter can be implemented using either *filter* or *convolve*, but *convolve* is strongly preferred due to its much greater computational efficiency. The impulse response is also available as an optional output to stdout. To see the frequency response, type

fastfir [flags] [filter_file] | spect -f

or

impulse 1024 512 | convolve filter_file | spect -f

The basic idea behind the design is that an ideal filter has an infinitely sharp transition from passband to stopband. As a consequence, it also has an infinitely long impulse response. Simply taking a finite number of samples of the ideal impulse response does not produce a very good filter. But multiplying the finite impulse response by a smoothly varying window can produce a surprisingly good filter. Hence, the filter design reduces to four simple steps:

1) Specify the number of samples in the impulse response via the -n flag.

2) Specify the filter type (i.e., lowpass, highpass, bandpass, or bandstop) via the -x flag.

3) Specify the window type (e.g., hamming, kaiser, etc.) via the -w flag.

4) Specify the cutoff frequency(s) via the -f flag. If the filter is a passband or stopband

filter, then two -f specifications are required: one for the lower edge of the pass(stop) band and one for the upper edge. It is standard to state these values as decimal fractions of the sample rate (e.g., .15), but they may also be expressed as actual frequencies provided that the -R flag specifies the intended sample rate.

For a more detailed discussion of these four steps, the reader is strongly encouraged to consult the associated helpfile for this program. Another useful reference may be the *Programs for Digital Signal Processing*, published by the IEEE. This program is essentially a direct translation of the program WINDOW in that book.

FILES

AUTHOR

Mark Dolson

SEE ALSO

impulse(1carl), filter(1carl), convolve(1carl)

BUGS

NAME

fc - floatsam counter

SYNOPSIS

fc < floatsams > count

DESCRIPTION

fc reads floatsams (32-bit binary floating point samples) on its standard input, and when exhausted, prints the number read on its standard output.

If it receives a SIGKILL signal (via the delete key, usually), it prints out the number of floatsams counted so far, and exits.

NAME

fftanal – analyze power spectrum of fft for peaks

SYNOPSIS

fftanal [**flags**] < fft_power_spectrum > output

Input must be a file or pipe of floatsams. Output is [frequency, amplitude] pairs, arabic if standard output is a terminal, floatsams otherwise.

FLAGS

-nN number of expected nodes (1024)

-wN window size used by the fft (16K)

-RN sample rate of original sample data (49152)

-tN minimum amplitude threshold (.0001)

DESCRIPTION

fftanal reads the power spectrum on its standard input (from, for example, *spect(1carl)*) and walks it through looking for peaks which exceed the threshold. It produces a data-reduced stream of numbers of the [x,y] values of the peaks. The threshold may be specified with the "dB" postoperator. For terminal output, amplitude is in dB of intensity.

AUTHOR

Gareth Loy

SEE ALSO

spect(1carl), harmanal(1carl).

NAME

filter – program for performing FIR or IIR digital filtering

SYNOPSIS

filter filter_file < floatsams > floatsams

DESCRIPTION

The **filter** program is for FIR and IIR digital filtering. You provide a filterfile such as designed by *fir* or *lpc*, and an input signal. The input and output data streams must be floatsams (binary, floating-point sample values). This program will work with any legal filter, but it is best used on IIR filters and on FIR filters with short (e.g., less than 20 samples) impulse responses. For long FIR filters, the **fastfilt** filtering program is more efficient.

Note: the gain of the filter will depend on the specified filter_file. If this file is not normalized to unity gain (e.g., filters produced by *lpc*), then an additional gain stage may be required to keep the peak value under 1.0.

AUTHOR

Mark Dolson

SEE ALSO

fir(1carl), lpc(1carl)

BUGS

NAME

fir – IEEE program for optimal design of FIR filters

SYNOPSIS

fir

DESCRIPTION

This program designs FIR (finite impulse response) filters according to an optimal algorithm. Unfortunately, it is not at all easy to use. Some helpful suggestions are contained in the helpfile for this program, but it is still not recommended for the novice. A good, but non-trivial, reference is *Programs for Digital Signal Processing*, published by the IEEE, or the book *Theory and Application of Digital Signal Processing* by Rabiner and Gold. Also, you can get all but the first parameter to take on the values required for a simple lowpass filter by typing carriage returns to the prompts.

After design is complete, you are asked for a filterfile name. This file is written out in filterfile format so that the **filter** and **fastfilt** programs can use it. If the filter impulse response is more than 20 samples long, then the **fastfilt** program is strongly preferred. The filterfile is an ASCII file and is intended to be self-documenting.

FILES

AUTHOR

J. O. Smith and F. R. Moore

SEE ALSO

impulse(1carl), filter(1carl), fastfilt(1carl)

BUGS

Some input parameters specify impossible filters, or ones which cause numerical difficulties. These cases can cause irritating problems. For example, specifying an even number of coefficients and zero gain at DC (e.g., a highpass filter) results in an extremely poor design. This is because the program is wired to produce symmetric impulse responses for all multiband filters. But this is one case in which an antisymmetric response would be more appropriate. This could easily be incorporated into the program, but a simpler solution is not to specify even numbers. Odd numbers also have the advantage of leading to an integral sample delay. Lastly, this program should be rewritten to be kinder to its users; but that would be a thankless chore!

NAME

floatsam – floatsam sample data, and headers

DESCRIPTION

Data

CARL implements sample data streaming between processes in the following way. Where two processes generate/receive sample data, it is piped as C **float** class data. These are called *floatsams*. Where a process that generates sample data is directed to a terminal, arabic, or ASCII (depending on what you like to call them) numbers are printed. An alternate data format is C **short** class data. These are called *shortsams*. This corresponds to the format suitable for conversion through DACs and ADCs.

Headers

A header consists of a stream of ASCII character codes. A sentinel at the beginning of a floatsam stream (consisting of the character string "HEAD") signals the beginning of a header. Header elements are called *properties* of the floatsam stream, and they consist of pairs of NULL-terminated (ASCII 0) strings. The first of a pair is taken as a *name* and the second as a *value*. A floatsam stream header consists of property lists, and looks as follows:

HEAD '\0' <revision_level> '\0'
<name1> '\0' <value1> '\0'
<name2> '\0' <value2> '\0'
<nameN> '\0' <valueN> '\0'
TAIL '\0' <revision_level> '\0'

After the very last NULL of the very last string of the header, as many NULLs must be padded to align the stream to read/write the next sample correctly (modulo **sizeof(float)** or **sizeof(short)**, where appropriate.

Vocabulary of standard properties

Truely general properties only should be in the system vocabulary list, kept in the include file <carl/defaults.h>. They should be all capitalized to distinguish them from local vocabulary that might be invented between a subset of cooperating processes which wish to communicate special information, which should be lower-case, or mixed case. Such special vocabulary terms should possibly include a key word or letter to indicate the special vocabulary to which it belongs. Refer to <carl/defaults.h> for the current list.

FILES

<carl/procom.h>, <carl/defaults.h>.

SEE ALSO

getfloat(3carl), floatsam(3carl), "Procom - Interprocess Sample Data Communications Facility for UNIX", CARL Technical Report.

NAME

floatsav – iteratively save floatsams on a dynamic array

SYNOPSIS

```
#include <carl/carl.h>

floatsav(base, floatsam)
float **base, floatsam;

reinitfloatsav()
```

DESCRIPTION

floatsav() comes in handy when reading an input floatsam (32-bit floating point data) stream where you want to save it in an array. **floatsav**() will take one float at a time and automatically scale the size of the buffer to fit all data, by using *malloc(3)* and *realloc(3)*. Its first argument should be the address of a pointer to a float. **floatsav**() will deposit the address of the base of the array it claims in this pointer. The value of the pointer must be initialized to 0. The second argument is the floatsam to be saved in the array. Iterative calls to **floatsav**() with the same base address will save subsequent floatsams on the same array. **floatsav**() returns the number of floatsams saved since the last call to **reinitfloatsav**(). **reinitfloatsav**() resets the count to 0, and sets up **floatsav**() to start a new array on its next call. It is only necessary to call **reinitfloatsav**() after the first array is built.

AUTHOR

Gareth Loy

FILES

/usr/local/lib/libcarl.a.

SEE ALSO

gen0(1carl).

NAME

gain – scales a digital sound signal

SYNOPSIS

gain factor < floatsams > floatsams

DESCRIPTION

gain reads floatsams (32 bit binary floating point numbers) from stdin, scales them by the gain *factor* supplied as its argument, and writes the scaled values on stdout. Factor may be an expression following the expression rules given for *expr(3carl)*. If the output is connected to a terminal, ASCII data is produced. *gain* is intended to be included in pipes with programs such as *para, chan, sndin, cmusic, impulse,* etc. The following example generates an impulse signal 50 samples long, consisting of .707, followed by 49 zero values: impulse 50 |gain -3dB

AUTHOR

F. R. Moore

SEE ALSO

newwire(1carl), para(1carl), chan(1carl), impulse(1carl), getfloat(3carl)

NAME

gaindelay - simple inline program to scale and delay a digital signal

SYNOPSIS

gaindelay gain samples_delay < floatsams > floatsams

DESCRIPTION

gaindelay reads floatsams from stdin, scales them by the gain factor supplied as its first argument, delays them by the number of samples specified by its second argument, and writes the scaled, delayed values on stdout. It is intended to be included in pipes with programs such as para, chan, sndin, cmusic, impulse, etc. The following example generates an impulse signal 54 samples long, consisting of 4 zero values, followed by .707, followed by 49 zero values:

```
impulse 50 |gaindelay -3dB 4
```

AUTHOR

F. R. Moore

NAME

gen0 gen1 gen2 gen3 gen4 gen5 gen6 - cmusic function generators

SYNOPSIS

genN – LN [**flags**] specification > floatsams
where -LN specifies N floatsams of output.

DESCRIPTION

These programs are the set of **gen** commands published with *cmusic(1carl)*. They are utilized in *cmusic* to generate waveform, envelope, and other control functions. They are actually standalone programs which can be run "as is" to create arbitrary functions for other purposes. In addition to *cmusic*, for instance, *Player(1carl)* provides an interface to utilize these programs. There are other **gen** type commands that can be used in the context of a *cmusic* score. Refer to the SEE ALSO section below.

This manual page shows their use as standalone programs. Refer to *cmusic(1carl)* for their use in that context.

All programs except **gen0** require a **-LN** flag which specifies the length of the function to create.

In addition, some of these programs take other optional flags. Two flags in particular, -o and -c determine whether the function created is "open" or "closed". An open function is usually required for periodic functions designed for iterative use, as in oscillators which read a function many times in succession. In this case, the last point of the function does not go full cycle, but stops one sample short, so that when the oscillator wraps around to the beginning again, it experiences no discontinuity, and therefor produces no click.

A closed function is used e.g. for amplitude control. Here one typically wants the last value equal to the first. If, for instance, a trapezoid envelope function were generated in open form, the last point would not be quite zero, and there would likely be a click at the end of a note.

The programs that allow open/closed function control specify this with – o and – c flags.

GEN0 - Normalize a function

gen0 [**-LN**] [**max**] < floatsams > floatsams

Gen0 scales the function it reads on its standard input so that the maximum value of the function is equal to **max**. If **max** is omitted, it is assumed to be 1.0. **gen0** will insure that no value exceeds the absolute value of max in either case. For this *gen* statement only, the length flag may be omitted, in which case *gen0* reads its standard input until it is exhausted. If the length flag is supplied, *gen0* stops reading input after that many samples are read.

Note that *cmusic* implements **gen0** internally. This standalone program simulates its behavior. Because of the nature of the algorithm, **gen0** must read up the entire function before writing it out. Because of this, care should be made not to use it to normalize extremely large data streams.

GEN1 - straight line segment function

gen1 -LN t1 v1 t2 v2 . . . tN vN > floatsams

gen1 generates a (closed) function which starts with value v1 at point t1 (the beginning of the function), continues in a straight line to value v2 at point t2, etc., until final value vN is reached at the end of the function. Example:

gen1 -L1024 0 0 1/3 1 2/3 1 1 0 > file

This generates a trapezoidal function; straight line from 0 to 1 for the first third of the function, level value of 1 for the middle third, falling from 1 to 0 in the final third.

GEN2 - Fourier synthesis function generator

gen2 -LN [-o (default) or -c] a1 a2 ... aN b0 b1 ... bM N > floatsams

aK is the amplitude of harmonic K in sine phase, (the fundamental frequency, corresponding to a full period of the function, corresponds to K = 1), and bK is the amplitude of harmonic K in cosine phase. Note that the first cosine component is at 0 Hz ("D.C."). The shape of the final (open) function is determined by the sum of all the components specified; the peak amplitude of the final function is normalized to 1.0. The use of gen2 is deprecated. Use gen5 whenever you wish to understand what you are doing. Examples:

(sine wave)

```
gen2 -L32 1  1
```

(cosine wave)

```
gen2 -L32 0 1  0
```

(the first five components of a square wave)

```
gen2 -L4096 1 0 1/3 0 1/5  5
```

(same as in the previous example with a .5 D.C. offset)

```
gen 0 gen2 f4 1 0 1/3 0 1/5 .5 5
```

GEN3 - simple line segment function

gen3 -LN v1 v2 ... vN > floatsams

The arbitrarily long list of values v1, ..., vN specifies relative amplitudes at equally spaced points along a (closed) function. Thus

```
gen3 -L32  0 1 1 0
```

specifies a trapezoidal function which has 3 parts: the first rises in a straight line from 0 to 1, the second is steady at 1, and the third falls from 1 to 0. If all values are positive, the function is scaled to be only positive. All negative values result in an all negative function. Values may be both positive and negative, resulting in a function which ranges between + 1 and -1.

GEN4 - exponential curve segment generator

gen4 -LN t1 v1 x1 t2 v2 x2 ... tN vN > floatsams

The t values specify positions along the horizontal axis on an arbitrary scale (as in **gen1**), the v values specify values of the (closed) function at these points, and the x values specify transitions from one point to another. x = 0 will yield a straight line, x < 0 will yield an exponential transition, and x > 0 will yield a logarithmic transition. If segment[J,i] is the i-th function value in the transition from v[J] to v[J+1], then its shape is determined by the formula:

segment[J,i] = v[J] + (v[J+1] - v[J])*(1 - exp(i*x[J]/(N-1)))/(1 - exp(x[J]))

for 0 <= i < N, where N is the number of function points between t[J] and the next horizontal value. No x value is given after the final point. Examples:

(straight line transitions)

```
gen4 -L32  0 0 0  1/3 1 0  2/3 1 0  1 0
```

(exponential transitions)

```
gen4 -L32  0 0 -1  1/3 1 0  2/3 1 -1  1 0
```

(similar to last example but with quicker exponential transitions)

```
gen4 -L32  0 0 -5  1/3 1 0  2/3 1 -5  1 0
```

The x value specifies the number of exponential time constants between the endpoints of a transition. A small negative value specifies a curve not very different from a straight line, but which gets near the second value more quickly. A large negative value is more curved, and approaches the second value more quickly. The latter kind of shape is very useful for specifying very sharp attacks while avoiding clicks, for example.

GEN5 - Fourier synthesis generator

gen5 -LN h1 a1 p1 h2 a2 p2 ... hN aN pN > floatsams

Each Fourier component of the (open) function is described by a triplet specifying a harmonic number, an amplitude, and a phase offset relative to sine phase. As many components may be supplied as desired, but all three values must be supplied for each component. Phase angles are

in radians (cf. "Deg" post operator in expressions); harmonic numbers need not be integers. Examples:

(a sine wave)
gen5 -L32 1 1 0
(first 3 components square wave)
gen5 -L32 1 1 0 3 1/3 0 5 1/5 0
(raised cosine wave)
gen5 -L32 1 -.5 90Deg 0 .5 90Deg

GEN6 - random table generator

gen6 -LN > floatsams

Gen6 fills the function named with random samples in the range -1 to + 1.

AUTHOR

F. R. Moore

SEE ALSO

genraw(1carl), step(1carl), cspline(1carl), chubby(1carl), quad(1carl).

BUGS

There are some differences between running gen programs within *cmusic* and running them standalone. Unlike *cmusic*, commas should not appear to separate arguments on the command line. The postoperators that relate to sampling rate are not available, such as Hz and sec.

NAME

genraw – read raw floating point functions into cmusic

SYNOPSIS

genraw – LN function_file

– LN is for example, -L1024 for a 1024-point function.

DESCRIPTION

genraw is an implementation of a "gen" function call for *cmusic(1carl)*. The action of genraw is to read the named file, which must contain binary floating point numbers (called *floatsams* at CARL). The floatsams are then written on its standard output. If called from inside *cmusic*, this will cause the contents of the file to be copied into a function in *cmusic*. In this way one can use, for example, results of the analysis of natural sounds as waveforms or envelopes in *cmusic*.

Here is a sample call from a shell:

```
% genraw -L1024 filename
```

and a sample statement in a cmusic score:

```
var 0 s1 "filename";
gen 0 genraw f1 s1;
```

genraw will force the number of floatsams it writes to its standard output to equal the number specified in – *LN* by linearly interpolating the number of floatsams in the input file to stretch or shrink the file to fit. This guarantees that *cmusic* will get as many points as it wants, but may have side effects of altering the character of the function if the stretching or shrinking are extreme.

If standard output is a terminal, arabic number values of results are printed, if a file or pipe, floatsams are written.

SEE ALSO

cmusic(1carl)

BUGS

cmusic gen functions must accept -c and -o flags for "closed" and "open" mode. *genraw* only produces -o format functions.

Linear interpolation is not an ideal way to stretch or shrink a file to fit in a function. *cmusic* provides mechanisms to alter the size of a function. The best fit would make the size of the function equal to the number of floatsams in the file. If you must compress the data to get it down to a reasonable length, try doing it with *srconv(1carl)*.

NAME

getfloat, fgetfloat, putfloat, fputfloat, flushfloat, fflushfloat, getshort, fgetshort, putshort, fputshort, flushshort, fflushshort, fgetfbuf, fputfbuf, fgetsbuf, fputsbuf – read and write sound sample streams from UNIX files or pipes

SYNOPSIS

```
#include <stdio.h> #include <carl/carl.h>

int getfloat(floatptr)
float *floatptr;

int fgetfloat(floatptr, iop)
float *floatptr;
FILE *iop;

int putfloat(floatptr)
float *floatptr;

int fputfloat(floatptr, iop)
float *floatptr;
FILE *iop;

int flushfloat()

int fflushfloat(iop)
FILE *iop;

int getshort(shortptr)
short *shortptr;

int fgetshort(shortptr, iop)
short *shortptr;
FILE *iop;

int putshort(shortptr)
short *shortptr;

int fputshort(shortptr, iop)
short *shortptr;
FILE *iop;

int flushshort()

int fflushshort(iop)
FILE *iop;

fgetfbuf(fp, n, iop)
float *fp;
short n;
FILE *iop;

fputfbuf(fp, n, iop)
float *fp;
```

```
short n;
FILE *iop;

fgetsbuf(sp, n, iop)
short *fp;
short n;
FILE *iop;

fputsbuf(sp, n, iop)
short *fp;
short n;
FILE *iop;
```

DESCRIPTION

These routines form the core of conventional data i/o routines used by all CARL programs. They work in a way analogous to getchar(3) and putchar(3). They return a positive value if successful, 0 on EOF (all the *get* routines), and a negative value on error.

Sample data is transmitted between programs as binary floating point samples called *floatsams*. *Shortsams* are binary short integers. The term *floatsam* is used generically on occasion to mean both kinds. Most programs only deal with floatsams, except those programs that specifically say they read both, or that convert from one format to another.

THE ROUTINES

getfloat() reads a single floatsam from the standard input. Note, that the address of the float must be passed to **getfloat()**. **getshort()** reads a single shortsam from the standard input. These are both actually macros for the following functions where iop is defined as stdin.

fgetfloat() and **fgetshort()** read from the file opened on iop.

putfloat() and **putshort()** take the address of a float/short-sam to write to stdout. They are macros for **fputfloat()** and **fputshort()** with iop defined as stdout.

flushfloat() and **flushshort()** MUST be called before exiting programs that have used **putfloat()** or **putshort()** in order that the last buffer of data be written on the file. These are macros for **fflushfloat()** and **fflushshort()** with iop defined as stdout.

fgetfbuf() and **fgetsbuf()** read buffered blocks of floatsams and shortsams respectively. **fputfbuf()** and **fputsbuf()** write buffered blocks of floatsams and shortsams respectively.

HEADERS

All these routines are able to detect the presence of headers on the data as defined in headers(5carl). The headers are stripped off, and set aside where the header routines described in headers(3carl) can find them.

FILE

(at CARL) /usr/local/lib/libcarl.a.

AUTHOR

F. R. Moore conceived getfloat() and putfloat(). Gareth Loy implemented the full isomorphic set, and added headers.

SEE ALSO

newwire(1carl), floatsam(5carl), procom(3carl).

NAME

glitch - one sample error detection

SYNOPSIS

glitch [-Nnumber] < floatsams > error_histogram

DESCRIPTION

glitch reads floatsams (32-bit binary floating point samples) on its standard input and when the input is exhausted, prints a histogram of the N likeliest samples to produce a click in playback. (The default is N = 22).

glitch works by taking the slope of two samples to predict by linear extrapolation where the next one ought to lie. Since musical signals are generally not linear, the estimate will have some error. The magnitude of this error is compared to previous errors, and the 22 largest errors and their sample indices are saved and reported. The errors are reported in decreasing order of magnitude except that some effort is made to group together errors with adjacent indicies.

Some insight into the glitch detection algorithm can be gained by considering several idealized kinds of clicks and noting that the algorithm is simply looking for large differences in slope.

The first case to consider is that of a sudden change in slope such as might result from butting two signals together in such a way that the instantaneous levels are matched, but not the slopes; this would show up as a single large error value with the index being the sample number at which the change in slope occurred.

A second case which also might result from butting two signals together is a sudden change in actual value from one sample to the next; this would show up as two adjacent error values, one positive and the other negative.

A third case is that of a single erroneous sample value amidst a stream of correct values; this would produce three large error values in succession, with one positive and the other two negative or vise versa.

In practice, some care is necessary in order that the algorithm produce useful results. First, the user should be sure that the signal under examination is monaural - a stereo file will hopelessly confuse things! Second, the user should narrow down the region to be examined as much as possible (e.g., by using interactive play). This is particularly true when trying to detect more than one glitch in a single soundfile; small glitches will almost always be hidden by larger ones if the algorithm is applied to the entire file at once.

It is important to remember that it is the *error* signal that is being reported, not sample amplitudes. The error signal is the difference between the expected and observed samples.

glitch also prints the simple average of the error signal measured for each sample. Comparing the maximum error to the average helps estimate what is reasonable, and what is out of line.

BUGS

AUTHOR

Gareth Loy and Mark Dolson.

NAME

help - get help about CARL programs

SYNOPSIS

help
Prints the main list of topics (files) and supertopics (directories).

help X
If X is a topic file, its contents are printed.
If X is a supertopic directory, its contents (subtopics) are printed.

help -1
prints 1-line summary of the main topics and supertopics.

help -1 X
If X is a topic file, its 1-line summary is printed.
If X is a supertopic directory, a 1-line summary of each of its subtopics is printed.

help -a X
If X is a supertopic, recursively lists all subtopics under X.

help -a
Recursive lists ALL topics and subtopics.

help -s WORD
Search through all help files and list those which mention "WORD".

NOTE: X may be a simple topic or supertopic name, or several supertopics separated by slashes (/), ending with either a topic or a supertopic name. For example: 'help cmusic', 'help -1 cmusic/gens', 'help cmusic/gens/gen0'.

DESCRIPTION

help prints online documentation for CARL programs. This documentation is collected in a special directory for help files. All users with help-ful information are invited to add help files (say 'help helper' for a description of how to do this).

FILES

(at CARL) /usr/local/lib/help.

AUTHOR:

F. R. Moore

NAME

hist – calculates histogram of sound signals

SYNOPSIS

hist < floatsams > histogram (ASCII display)
or
hist – e < floatsams > floatsams, and > & histogram

DESCRIPTION

hist reads samples from stdin and calculates a histogram of the amplitude values contained in that signal. The input data stream must be floatsams. In the first usage given above, it prints this histogram and various amplitude measures after all samples have been read on stdout. In the second usage, when the -e flag is given, the histogram is written to stderr, and the input samples are copied to stdout. This allows hist to be inserted in a sample data stream as a "test probe."

The following examples determine a histogram for sound file boom:

```
sndin boom |hist
sndin boom |hist -e |sndout boom2 (histogram placed on stderr)
```

AUTHOR

F. R. Moore

SEE ALSO

rms(1carl), peak(1carl), getfloat(3carl)

NAME

ichan – multiplexed sound signal processing with parallel pipes

SYNOPSIS

ichan #_of_parallel_channels process_1
... process_N < floatsams

DESCRIPTION

ichan provides multichannel signal processing by demultiplexing its standard input, feeding each channel (through pipes) to one of several monophonic processing pipelines which run as parallel processes.

Unlike *chan*, **ichan** does not remultiplex and produce any output.

```
cmusic stereo.sc |ichan 2 "peak"
```

takes the output of cmusic, separates the 2 channels, applies "peak" to each channel, which produces two reports in parallel.

AUTHOR

F. R. Moore

SEE ALSO

chan(1carl), para(1carl), newwire(1carl), getfloat(3carl)

NAME

impulse – generates an impulse signal

SYNOPSIS

impulse [length] [delay] > output
Default length = = 4096. Default delay = = 0.

DESCRIPTION

impulse is a simple command for generating a digital impulse signal:

```
impulse 100
```

will generate a floating 1.0 followed by 99 0.0's at its standard output.

```
impulse 100 10
```

will generate 10 floating 0.0's followed by a 1.0 followed by 89 0.0's at its standard output. *output* normally consists of binary, floating-point sample numbers. If the output is connected to a terminal, ASCII data is produced.

The following example would generate the impulse response of a filter called test.flt, and store it on file imp.resp:

```
impulse 1000 |filter test.flt > imp.resp
```

AUTHOR

F. R. Moore

SEE ALSO

newwire(1carl), getfloat(3carl)

NAME

ispell - spelling error fixer

SYNOPSIS

ispell text_file

interactive options:

look:	l < key>,
grep:	g < key>,
sed:	s < script>,
new file:	n < filename>,
help:	h,
print words:	p,
shell:	! < command>,
print current file name:	%,

all args to *look*, *grep*, and *sed* are double-quoted.

SHORT DESCRIPTION

text_file is first fed to *spell(1)*, and a list of words not in the dictionary is printed. Then a '*' prompt appears. Type 'h' at that point for help on interactive commands which are available to help correct spelling errors.

LONG DESCRIPTION

ispell is a program that lets you conveniently run various existing system programs to interactively correct spelling errors. The programs **ispell** knows about are *spell*, *look*, *grep* and *sed*. An interactive command to **ispell** consists of a letter (which is expanded into the full name of the system command to run) followed by statements to be passed as arguments to the system program.

When invoked, **ispell** first runs *spell* to generate a list of words not found in the dictionary. It displays them, then shows a '*' prompt. Either 'h' or '?' displays a help message. Here are examples of all the commands:

look: **l < key>**

The command

*l fort

will search the UNIX dictionary all words beginning with "fort" to be printed on the screen.

grep: **g < key>**

The command

*g aver

will print out all lines in the file containing this string.

sed: **s < script>**

The command

*s s/explain/explane/g

will change all of the former spellings in the file to the latter.

new file: **n < filename>**

The command

*n nextfile

will cause nextfile to be sent through spell. It then becomes the object of **ispell's** interactive commands.

help: **h, ?**
prints a short help message.

print words: **p**
redisplays all words found by spell (does not reflect changes).

shell: **! <command>**
The prefix '!', as in most other unix programs, escapes the following command to a subshell. Note there must be a space between the ! and the command.

print current file name: %
This reveals the name of the current file being corrected.

The command line sent to *grep* and *sed* is automatically double-quoted to keep shell meta-characters such as '*' and friends from screwing things up.

AUTHOR
Gareth Loy

SEE ALSO
spell(1), look(1), sed(1), grep(1)

BUGS
No syntax checking is done on the command being passed to *sed*, and *grep*. If it is not well formed, strange things can happen to your file. It is worthwhile to run **ispell** on a copy of your original text file for safety.

NAME

janus – smooth the attack and decay of a sound

SYNOPSIS

janus [**-bN**] [**-eN**] [**-RN**] [**-h**] < floatsams > floatsams

FLAGS

-b scale beginning until time N (default= .05sec)

-e start ending scaling at time N from end of file (.05sec)

-R override default sampling rate for time calculations (value obtained from header, else default= DHISR Hz)

-h prints a terse usage message

DESCRIPTION

The purpose of **janus** is for smoothing out the edges of a sound that may have discontinuities which cause clicks or other undesirable noise at the beginning and/or end of a sound.

The beginning and ending of the sound is amplitude-scaled by a portion of a sine function over the duration of time specified with the -b and -e flags.

For the beginning of the file, the scaling function used is

$$Yn = Xn * 0.5 * (\cos(\pi * i / Nb + \pi) + 1.0)$$

where Nb is the number of samples to be scaled at the beginning of the sound, and i is the current sample within that range. For the ending of the file, the scaling function is

$$Yn = Xn * 0.5 * (\sin(\pi * i / Ne + \pi/2.0) + 1.0))$$

where Ne is the number of samples to be scaled at the end, and i is again the current sample within that range.

In between the beginning and ending scaling, the transform is simply

$$Yn = Xn$$

Sample rate is set from the value obtained from the header, if any, else the -R flag value is used, else the compiled-in default, DHISR is used.

AUTHOR

Gareth Loy

NAME

libasw – audio switch daemon library routines

SYNOPSIS

```
int openasw()
int closeasw(fd)
int fd;

char *connasw(name, chans)
char *name;
short *chans;

char *disconnasw()

char *resetasw()
```

DESCRIPTION

These routines are the C interface to the CARL audio switch daemon. All routines, except for **openasw** and **closeasw**, return a status message from the daemon. The string "ok" means that the connection was successfully made; other strings are error messages.

openasw returns -1 on error or a socket descriptor upon success. **closeasw** is used to shutdown the socket returned by **openasw**. It returns -1 on error or 0 upon success. Both **openasw** and **closeasw** are called automatically by all of the other routines and normally isn't needed; they are provided for users that are doing unnatural things and need to send messages to the daemon directly.

connasw takes two arguments. The first is the name of the device to connect to, the second is a list of channels to connect. These channels of the output device are connected to your space.

disconnasw takes no arguments. It breaks all connections to your space.

resetasw takes no arguments. It breaks all connections to all spaces.

The audio switch daemon at CARL is configured so that each terminal corresponds to an input space; the terminal name is used for the symbolic name of that input space. At CARL we also have some SUN workstations. For their symbolic name their network machine name is used. The routines **connasw** and **disconnasw** call a internal routine that determines the symbolic name of your input space by looking in the */etc/utmp* file.

AUTHOR

Rusty Wright

SEE ALSO

asw(C), aswdaemon(C)

NAME

libieee - standard ieee digital signal processing subroutines

SYNOPSIS

```
#include <math.h>

double fourea_(x,&N,&isign)
        float x[2*N];
        int N, isign;

double fast_(x,&N)
        float x[N+2];
        int N;

double fsst_(x,&N)
        float x[N+2];
        int N;

double fft_(a,b,&one,&N,&one,&isign)
        float a[N], b[N];
        int N, isign, one = 1;

double auto_(&N,x,&M,a,&alpha,rc)
        float x[N], a[M+1], rc[M], alpha;
        int N, M;

double covar_(&N,x,&M,a,&alpha,grc)
        float x[N], a[M+1], grc[M], alpha;
        int N, M;
```

DESCRIPTION

The following descriptions are necessarily too brief to be complete. The recommended source in all cases is *Programs for Digital Signal Processing*, I.E.E.E. Press, 1979.

*fourea_* is primarily a demonstration of the FFT algorithm. It takes the FFT of the N complex values in x (where x[0] is the first real value, and x[1] is the first imaginary value, x[2] is the next real value, etc.).

INPUT: x contains the N complex values to be transformed; isign is -1 to specify a forward transform (time domain to frequency domain) and +1 to specify an inverse transform (frequency domain to time domain); N must be a power of 2.

OUTPUT: x contains the N complex values of the transform.

NOTE: Other FFT subroutines (listed below) are generally more appropriate.

*fast_* and *fsst_* are the most commonly used routines for taking forward *(fast_)* and inverse *(fsst_)* FFT's for REAL input sequences.

INPUT: On input to *fast_*, x contains the N-point (real) sequence to be transformed; N must be a power of 2.

OUTPUT: On output from *fast_*, x contains the (N/2 + 1) complex values of the transform (where x[0] is the DC value, x[1] is zero, x[2] is the next real value, x[3] is the next imaginary value, etc.). The remaining (N/2 - 1) complex values can be deduced from these because the input is real.

INPUT: On input to *fsst_*, x contains the (N/2 + 1) complex values of the transform; N must be a power of 2.

OUTPUT: On output from *fsst_*, x contains the N (real) values of the sequence.

*ff_* is a mixed radix FFT; this means that fairly efficient transforms can be performed for arbitrary values of N (i.e., N need not be a power of 2).

INPUT:a holds the real components of the data to be transformed, and b holds the imaginary components. N is the number of samples in a; it is also the size of the transform. isign is -1 for a forward transform and +1 for an inverse transform. one is simply a variable equal to 1.

OUTPUT: a holds the real components of the transform, and b holds the imaginary components.

*auto_* and *covar_* are the most commonly used routines for performing linear prediction analysis. They each calculate the M weighting coefficients which allow x(n) to be best estimated from its previous M values. This is equivalent to calculating the M coefficients of the IIR (allpole) filter which best fits the observed spectrum of the sequence x(n). *auto_* implements the autocorrelation method of linear prediction. It treats the data as though it is zero outside of the N values of x specified as input. *covar_* implements the covariance method of linear prediction. It makes no assumptions about the values of x beyond the N values specified as input. If N >> M, the results of both methods are very similar.

INPUT: x contains the N-point sequence to be analyzed; M specifies the number of coefficients to be used (up to a maximum of 20).

OUTPUT: The coefficients are contained in a[2] thru a[M+1] with a[1] = 1. A set of related coefficients is returned in rc[1] thru rc[M] (for *auto_)* and in grc[1] thru grc[M] (for *covar_)*. alpha returns an indication of how well the data fit the linear prediction model (A smaller alpha indicates better fit).

FILES

/usr/local/lib/libieee.

AUTHORS

Extracted from Programs For Digital Signal Processing, edited by the Digital Signal Processing Committee, IEEE Acoustics, Speech, and Signal Processing Society, IEEE Press, 1979.

SEE ALSO

libcarl.

DIAGNOSTICS

The most common error message is "floating exception (core dumped)" which almost always indicates that bad values have been passed to the subroutine being used.

NAME

arcsin, beta, cauchy, corrand, expn, frand, gamma, gauss, hyper, lin, logist, onefrand, plapla, randfi, randfc – stochastic functions

SYNOPSIS

```
#include <carl/libran.h>

double arcsin(rfunc)
        double (*rfunc)();
double beta(rfun, a, b)
        double (*rfun)();
        double a, b;
double cauchy(rfun, tau, lopt)
        double (*rfun)();
        double tau, lopt;
double corrand(lb, ub)
        double lb, ub;
extern double cor_factor;
double expn(rfun, delta)
        double (*rfun)();
        double delta;
double frand(lb, ub)
        double lb, ub;
double gamma(rfun, nu)
        double (*rfun)();
        double nu;
double gauss(rfun, sigma, xmu)
        double (*rfun)();
        double xmu, sigma;
double hyper(rfun, tau, xmu)
        double (*rfun)();
        double tau, xmu;
double lin(rfun, g)
        double (*rfun)();
        double g;
double logist(rfun, alpha, beta)
        double (*rfun)();
        double alpha, beta;
double onefrand(lb, ub)
        double lb, ub;
double plapla(rfun, tau, xmu)
        double (*rfun)();
        double tau, xmu;
double randfc(rfun, sRate, freq)
      double (*rfun)();
      double sRate, freq;
double randfi(rfun, sRate, freq)
      double (*rfun)();
      double sRate, freq;
```

DESCRIPTION

These subroutines implement a set of stochastic functions described as *cannons* by [Lorrain]. They are in two basic groups, noise *generators* and noise *modifiers*. The three generator routines are *frand()*, *corrand()*, and *onefrand()*. The *generator* subroutines take two arguments of a lower

and upper bound within which to generate their numbers.

All other subroutines are *modifiers* and take as their first argument a pointer to one of these *generator* subroutines. The *modifier* subroutines call the *generator* subroutines internally to get random values which they then transform according to their function. Some distribution functions such as *gauss()* will require several calls on their *generator* per call. Any source of random numbers may be substituted for the three basic *generator* subroutines. The substituted *generator* subroutine must obey the calling sequence:

```
double function_name(lower_bound, upper_bound)
    double lower_bound, upper_bound;
```

and provide return values between those ranges.

THE FUNCTIONS

In the description below, *Us* stands for the value obtained by the *modifier* subroutine from its *generator*.

Arcsin: arcsin(rfun) returns square(sin(pi * Us / 2)).

Beta: beta(rfun, a, b) produces various "U-shaped" distributions within the range [0,1]. In general, as a and b go from 1 to 0, the U gets deeper.

Cauchy:
cauchy(rfun, tau, iopt) returns Xs = tau * tan(pi * Us). If iopt= 1, the distribution is folded into the positive range only.

Correlated noise:
corrand(lb, ub) produces noise that goes from white to increasingly more correlated (to a dc constant) as the external double *cor_factor* goes from 0 to 1.

Exponential:
expn(rfun, delta) returns -log(Us) / delta.

Uniform noise:
frand(lb, ub)
returns a random number within the interval [lb,ub].

Gamma:
gamma(rfun, nu) produces an asymmetrical distribution that is 0 at x= 0, reaches its mode at nu - 1, with the mean at nu. The whole number value of nu is used. Nu must be greater than 0. Any fractional part is truncated.

Gaussian
gauss(rfun, sigma, xmu) Using sigma= 1.0, xmu= 0.0, an approximated standard gaussian distribution is produced.

Hyperbolic:
hyper(rfun, tau, xmu) returns tau * log(tan(pi*Us/2)) + xmu.

Linear: lin(rfun, g) returns g(1 - sqrt(Us).

Logistic:
logist(rfun, alpha, beta) returns (-beta - log(1/Us -1))/alpha.

1/f noise generator:
onefrand(lb, ub) returns a random number within the [lb,ub] interval according to the pseudo 1/f distribution algorithm described by Voss.

Laplace: plapla(rfun, tau, xmu) Cannon formula first selects Us in the range [0,2], then for Us > 1.0, it applies Xs = -tau * log(2.0 - Us) + xmu. For Us <= 1.0, it applies Xs = tau * log(Us) + xmu.

randf(rfun, sRate, freq),

linearly interpolating random function, produced at frequency **freq** of sampling rate **sRate**. A continuous function is generated by linearly interpolating between successive random values, where the frequency of new random values to interpolate between is set by **freq**. This is the equivalent of the RAN unit generator of MUSIC V.

randfc(rfun, sRate, freq),

cosine interpolating random function, produced at frequency **freq** of sampling rate **sRate**. Like randfi(), but it interpolates with the cosine function between 0 and pi. This produces a smoother effect in the transitions than **randfi()**, and its spectrum shows a smoother high frequency rolloff.

References

These routines are all used in *cannon(1CARL)*, and described in slightly more detail there. Note, there are some name differences between *cannon's* names and those of the subroutines here.

SEE ALSO

Martin Gardner, "Mathematical Games", Scientific American, March, 1979.
Denis Lorrain, "A Panoply of Stochastic 'Cannons'", Computer Music Journal, V. 4 #1, p. 53.

AUTHOR

Gareth Loy

NAME

libsf - subroutine library for csound file system.

SYNOPSIS

```
# include <carl/sndio.h>
float sfexpr(expression, sample_rate);
        char *expression;
        float sample_rate;
CSNDFILE *cpsfd(sound_file_descriptor);
        CSNDFILE *sound_file_descriptor;
float fsndi(sound_file_descriptor, sample_index);
        CSNDFILE *sound_file_descriptor;
      long sample_index;
CSNDFILE *accesf(name);
     char *name;
CSNDFILE *opensf(name, mode);
        char *name, *mode;
extern int sferror;
CSNDFILE *setsfd(sound_file_descriptor, ctrl, arg);
        CSNDFILE *sound_file_descriptor;
     char *ctrl, *arg;
closesf(sound_file_descriptor);
        CSNDFILE *sound_file_descriptor;
allclosesf();
```

DESCRIPTION

SFEXPR

sfexpr (expression, sample_rate) is an expression evaluator similar to that used in *cmusic(1carl)* but modified for figuring sample-rate dependent expressions, such as specifications of begin/end times in sound files. Rather than giving the full documentation here, only differences from **expr(3carl)** will be noted.

sfexpr() first evaluates the expression without reference to postoperators, then multiplies the result of the expression by the **sampling_rate**, and then applys the postoperators. The typical usage of **sfexpr()** is for users to supply expressions in seconds, and **sfexpr()** converts this to sample indices at the specified sampling rate.

Prefix – (minus).

Infix + , – , *, /, ^, %. All prefix and infix operators are applied to the expression first, then the result is multiplied by the **sampling_rate**, and then postoperators are interpreted.

Postoperators:

k, m, s, ms, S, where **dB** converts to dB of amplitude (e.g., -6dB is approximately equal to .5), **K** multiplies by 1024, **k** multiplies by 1000, **m** (minutes) multiplies by 60, **s** (seconds - default postoperator) multiplies by 1, **ms** (milliseconds) multiplies by .001, **S** (samples) divides by the **sampling_rate** (which recovers the result as a sample index expression).

The upshot of this is that all time expressions except those in which the **S** postoperator appears are taken to be times in seconds, and are multiplied by the sampling rate to get a sample index. Where the **S** postoperator appears, the expression is considered to be a sample index already, and the multiplication by the sampling rate is nullified.

Note: by setting the value of **sampling_rate** to 1.0, **sfexpr()** can be used to compute the natural value of expressions.

CPSFD

cpsfd (sound_file_descriptor) returns a pointer to a copy of the **sound_file_descriptor** passed to it.

FSNDI

fsndi (sound_file_descriptor, sample_index) returns the sample value of the sound file pointed to by **sound_file_descriptor** at the location specified by **sample_index.**

ACCESF

accesf (name) returns 0 if the sound file mentioned in **name** exists and the directory path can be traversed by this process, otherwise it returns -1.

OPENSF

opensf (name, mode) opens the named sound file. The **mode** argument takes a string that obeys command-line flag syntax to specify reading, writing, or both. If writing only, the file is created; any existing file is deleted. Additional flags then specify the properties of the new sound file. The file is opened by calling the program *opensf(1carl)* via **popen(2).** Please refer to *opensf(1carl)* for a description of the mode flags. If the open is successful, a pointer to a **CSNDFILE** structure is returned, else NULL. The value of the global variable **sfexpr** will be set to the value of any error detected.

FILE MODES AND ALLOCATION

New files are created by default as *non-realtime*. This simply means that the system does not need to fit the file into one contiguous block. No space is claimed for a new file so created until the first attempt to put something in it. A new file will be made *realtime* by mentioning any of the following flags:

-tn sets file to realtime and claims 1 cylinder by default;

-TN sets file to realtime and claims enough cylinders for time in seconds requested.

-CN sets file to realtime and N cylinders.

SETSFD

setsfd (sfd, ctrl, arg) sets a field on a **CSNDFILE** sound file descriptor structure. It returns a pointer to the modified structure. If **sfd** is NULL, a new empty sound file descriptor is created, and its address is returned. The **ctrl** string specifies the field to be modified, as described in **<carl/sndio.h>**, and **arg** is the data in string format to be substituted.

CLOSESF

closesf (sound_file_descriptor) takes a pointer to a sound file descriptor and closes it via **popen(3)** and *closesf(1carl).* **allclosesf()** closes all open sound files by walking a linked list of open sound files maintained by **opensf().**

SYSTEM LEVEL OPENSF AND CLOSESF

Two routines, **CSNDFILE sndesc *sopensf(name, mode, sfd) char *name; char *mode; CSNDFILE *sfd;** and **sclosesf(sfd) CSNDFILE *sfd;** are the *csound* system's privileged versions of **opensf()** and **closesf().** The difference is that these routines actually do the i/o to the sound file system disk and must therefor be executed in programs that are set-user-id disk, whereas **opensf()** and **closesf()** are front ends to programs *opensf(1carl)* and *closesf(1carl),* which in turn are set-user-id programs that call **sopensf()** and **sclosesf().** In any event, the following is a synopsis of **sopensf().**

Arguments:

name : filename spec, can be partial name

mode : string of following

r	read access
w	write access
rw	fully buffered read/write access

[rlw] F cause sdf file to be updated after each write
sfd : sound file descriptor
if NULL, use name to determine file
if !NULL, use filename on sfd to determine file and
if creating file (i.e., mode includes "w" and
file doesn't exist) use attributes of the sdf
to configure the file for nc, ns, fs, etc.
The new sfd will NOT have a pointer to the
same disk block (if any) as the sfd passed as argument.

FILES

These subroutines are in libcarl.a. At CARL, they are also in libsf.a.

AUTHOR

D. Gareth Loy

NAME

locksf, unlocksf - lock/unlock a sound file system

SYNOPSIS

locksf filesystem [message]

[The message part of the command is all arguments past the filesystem arg.]

unlocksf filesystem

filesystems at CARL:
/snd
/snd1

DESCRIPTION

locksf causes the named filesystem to be *masterlocked*. **unlocksf** does the inverse action.

This is the sound file system's version of single-user mode. When a filesystem is masterlocked, only members of a special group (the "disk" group at CARL) are able to access files on the locked filesystem. All other users receive a notice that that filesystem is locked. Only members of the disk group can masterlock/unlock a filesystem. Certain fatal errors in sound file programs can also cause the filesystem that the program was accessing to be masterlocked in order to keep continued access from making the filesystem unrecoverable.

The action of **locksf** and **unlocksf** is to manage a file in the root directory of each sound filesystem named MASTERLOCK. (E.g., at CARL, /snd/MASTERLOCK). The existence of this file signals a locked system. All sound file programs check the existence of this file automatically. If a program detects the existence of this file, it then checks to see if the user belongs to the disk superuser group or is root. If so, the program will print "lock: sesame." as a reminder to the superuser that the system is locked and to proceed with caution, and then allow the superuser to continue. If the person running the program is not in the superuser group, the message "The system has been MASTERLOCKED" is printed. It also prints the contents of the masterlock file. Fatal errors that create the masterlock file also put a diagnostic in it.

When running the **locksf** program, the 3rd. through nth. arguments are taken as a diagnostic message and placed in the masterlock file.

All programs that write the masterlock file do so in append mode, so as to enable catching a sequence of errors.

The **dumpsf** program can masterlock the filesystem during the time it is dumping.

FILES

<filesystem>/MASTERLOCK

AUTHOR

Gareth Loy

SEE ALSO

sndin(1csound), sndout(1csound), dumpsf(1csound).

DIAGNOSTICS

Attempts to run **locksf** or **unlocksf** by other than the superusers results in a "Sorry." message, and no action.

Running **locksf** by itself generates a terse usage message.

NAME

lookfor - notify when named person logs in

SYNOPSIS

lookfor login_name

DESCRIPTION

lookfor takes as its argument the name of the person you are looking for, and beeps your terminal when that person logs in. Run it in the background.

NAME

lpc - linear predictive coding of a signal spectrum with an allpole filter

SYNOPSIS

lpc [flags] < floatsams [> floatsams]

flags: (defaults in parenthesis)
-RN set input sample rate to N (usually read from stdin)
-bN begin at time N (0)
-wN set window size to N seconds (1024S)
-lM M pole linear prediction estimate via autocorrelation method
-kM M pole linear prediction estimate via covariance method

if filter_file is not specified the file tmp.flt will be created

(the optional output is the impulse response of the filter as
a stream of -wN floatsams to stdout)

DESCRIPTION

This program designs an IIR (all pole) filter which matches the spectrum of the input signal spectrum as closely as it can. Typically, the goal is to capture the broad resonances in the spectrum (e.g., the formants in speech) without including the individual peaks of each harmonic. The resulting filter can be used with the *filter* program to impose the linear prediction spectrum on some completely different signal. For example, the *lpc* program can be used to obtain a filter which matches the spectrum of a vowel sound. But note that this program produces only a single filter; "talking orchestra" effects can be obtained only with time-varying linear prediction.

Unfortunately, the inner workings of this program are rather tricky. Linear predictive coding is done on the basis that the spectrum matching filter is the inverse of the optimum linear predictor for the data. Another way of looking at this is that the program calculates the M weighting coefficients which allow the M'th input sample to be best estimated from the previous M values. However, this procedure is sensitive both to the value of M, to the value of N, and to the particular input samples available. For example, too large a value of M may result in the fine structure of the spectrum (i.e., individual harmonic peaks) being incorporated into the IIR filter. But this effect can be eliminated if N is exactly one period of the input. A general rule of thumb is that the order of the filter should be specified to be at least twice the number of major resonances present in the input spectrum.

There are two variations of *lpc* available. The covariance method is usually more accurate, especially for short data samples, because it does not force prediction of zeros outside the finite data sample. The autocorrelation method is equivalent for very long data samples and is faster than the covariance method.

To see the frequency response of the filter, type

sndin file |lpc [flags] |spect -f

or

impulse 1024 |filter filter_file |spect -f

A more detailed explanation and examples of typical usage are given in the helpfile for this program.

FILES

AUTHOR

Mark Dolson

SEE ALSO

fir(1carl), impulse(1carl), libieee(1carl), filter(1carl)

BUGS

Certain input specifications or input signals can cause either impossible filters to be specified, or may cause numerical difficulties. Either of these cases can be quite irritating.

NAME

lprev – configurable sound reverberator

SYNOPSIS

lprev [**-Dn**] [**-dN**] [**-gN**] [**-RN**] [**-TN**] [**-fX**] [**-CN**] [**-bN**] [**-c**] < floatsams > floatsams

FLAGS

Input and output must be a file or pipe. flags:

-D set dsig to N (0.3); sets ratio of direct/ reverb. signal

-dN set ring-time to N (4 sec. at prevailing sample rate)

-gN set reverb time coefficient to N (0.9) (must be < 1.0)

-RN set sample rate to N, derived from input header (DHISR Hz)

-TN set tap coefficient input scaler to N (1.0)

-fX specify configuration file X

-CN scale input to comb reverbs by N (1.0)

-bN set debug variable to N (0); values:

1 = print configuration on stdout,
2 = only output tap signal,
4 = only output comb filter signal,
8 = only output tap/comb mix.

-c print diagnostics about location of clipped signals.

All durations are in seconds. Use postop 'S' for sample times. Arguments may be expressions.

DESCRIPTION

lprev reads binary floating point sound sample data (floatsams), reverberates them, and writes the reverberated sound on its standard output, likewise as floatsams. It implements a reverberation scheme described by Andy Moorer ["About This Reverberation Business", *CMJ*, Vol. 3, #2.], which utilizes a tap delay line to simulate early reflections, and a section of comb filters with low-pass filters as feedback terms to simulate the decay of high frequency information through time because of air and wall absorption.

It does the "right thing" by default to act as a simple reverberator for any sound, to wit:

```
% sndin |lprev |sndout
```

but its main claim to glory is its modifiability, see below.

Flags

-R sets sampling rate N. If the input signal has a header and no -R flag is given, the rate is determined from the header, otherwise, the flag overrides the value determined by the header. If no flag or header occur, the rate defaults to DHISR. The sampling rate is computed in this manner before any delay lengths, or other time-dependent parameters.

-g This controls the total reverberation time. It defaults to 0.9, which seems to be about 3.5 seconds. A value of .87 for g provides a reverb time of about 1 second, The reverberation time can be estimated as

$$\frac{1-.366}{T}$$

where T is the reverberation time desired in seconds. A value of 0.98 gives a cathedral-like quality; 0.99 puts you in the Taj Mahal. Higher values, like .99999, are

possible but silly. The value must never equal or exceed 1.0, or the reverberator will become unstable.

-d sets the "ring-time" in seconds to N, that is, it says how long to continue running the reverberator after the input sound file has quit, allowing for the energy in the reverb to die away. It defaults to 4 seconds at the prevailing sampling rate, which is generally not enough, and should certainly be augmented for any value of -gN greater than 0.9.

-D sets the ratio of the direct/reverberated signal level to N. The default of 0.3 means that 30% of the signal put out will be direct signal, 70% will be the reverberated part. Adjust accordingly.

-T Because of its implementation, **lprev** cuts the overall level of the signal down somewhat. It can be scaled up or down with -T, where N is the gain scale, which defaults to 1.0. The coefficient N specified by -T is multiplied against the actual gain values used in the individual taps of the taped delay line. To adjust overall amplitude, take a loud section of your sound, reverberate it without -T, and pass it to *hist(1carl)*. *hist* might show that the peak reverberated signal is -18dB down, for instance; saying -T18dB scales the signal up by 18dB. Caution must be excersized, since it is possible that the signal will overflow if scaled up too far. Note, this is a relatively efficient way to scale the signal, and is preferable to rescaling the output with, for instance, *gain(1carl)*. This is because the -T flag affects a multiply which will be done in any case.

-fX specifies UNIX file X as a "configuration file", containing information about the number of unit reverberators and their delays and gains. With this facility, plus the flags, you can totally rearrange the performance of the reverberator. A configuration file consists of a series of text lines. Here is the syntax:

```
line_format   := = <spec_char> <blank> <coeff> <blank> <length> <'0>
  spec_char   := = t | c | a  (tap, comb, or allpass respectively)
  coeff       := = floating pt. number (between 0 and 1.0)
  length          := = floating pt. number (seconds delay length)
```

Lines may begin with a single letter, either 't', 'c' or 'a' for "tap", "comb" or "allpass". The next field is a gain associated with that unit, followed by a field giving the delay length in seconds at the prevailing sampling rate. Here are examples of single entries:

```
t     .134    0.0797
c     .26     .056
a     .7      .006
```

The number of 't', 'c' and 'a' entries determine the number of such sections; their lengths are specified in seconds; their coefficients are interpreted depending on what kind of unit is specified. Order of lines is irrelevant. Usually, 't' sections correspond to a peak in the impulse response in the first 50 - 100 ms. of a real room, where the gain is the normalized amplitude of the peak. For 'c' comb filter reverb sections, the gain terms adjust the rolloff of high frequencies through time. The delay lengths usually correspond to the periods between the 3 major parallel walls of the target room. The values of allpass are suggested to be a gain of .7 and a delay of 6 ms. Subsequent allpass sections are scaled by .9 for gain, and the nearest prime number of samples to .9 times the delay.

DIAGNOSTICS

Because this routine uses complex filtering and signal delay and mixing, it is difficult to normalize its behavior for all signals. The defaults have been chosen very conservatively so as to work

for most signals. The most likely cause of difficulty is clipping caused by summing variously delayed signals. The -c flag is useful to determine where the clipping is taking place. The following points are monitored for clipping, and are reported separately for each block of input samples processed by **lprev**.

Input signal;
The input signal has clipped. Attend to your input.

Tapped delay line output;
The sum of the tapped delay lines has clipped. Indicates that more than one tap has the same delay, or a tap gain is greater than 1.0.

Comb filter outputs;
The sum of the parallel comb filters has clipped.

Mixed tap and comb filter outputs;
The sum of the comb filters and tapped delay lines has clipped.

Allpass output;
The signal after the allpass filters has clipped.

Combined dsig and allpass;
The combined direct and reverberant signal has clipped.

To eliminated clipping, adjust the various internal gain controls for the different stages of **lprev**.

AUTHOR

Gareth Loy

SEE ALSO

"Lprev - Configurable Tap-delay/Low-pass Reverberator", in file /mnt/tutorials/lprev.dgl at CARL.

Andy Moorer, "About This Reverberation Business", *CMJ*, Vol. 3, #2.

NAME

lsf – list sound files, sound file directories

SYNOPSIS

lsf usage: lsf [flags] {[file][device][directory]} ...

flags:

-R recursively search directories
-l print long format
-f print sound file descriptor
-F print sound file descriptor with cylinder block list
-pX print only files with substring X in filename
-tX sort files according to date
X = c sort by creation date (default)
X = a sort by date last altered
X = r sort by date last referenced
-C sort files according to size in cylinders
-r reverse sort
-a print all files
-0..-9 print all files on selected device that should be dumped at that level

DESCRIPTION

lsf does for the *csound* file system what the regular UNIX **ls** command does for the UNIX file system. Without arguments, **lsf** lists all the files in your current sound file directory. This is usually your home directory, although it can be changed with *cdsf(1csound)*. If you specify a list of sound files, it will print the names of those that exist. If you specify a sound file directory, it will list all the files in that directory and names of any subdirectories. If you specify a device, it will recursively list all files and directories in all directories on that device.

Filenames may be specified as a full path, as a partial path, or as a name. A full path specifies the device, and any directories leading up to the name. An example command to have **lsf** verify that file *test* in the current sound file directory exists:

% lsf test

A partial file name, checking file *test* in user frm's home sound file directory:

% lsf /frm/test

A partial file name, checking file *test* in a subdirectory *foo* of the current sound file directory:

% lsf foo/test

A full filename, looking on device */snda* for user frm's file *test:*

% lsf /snda/frm/test

In each of these cases, a single file is listed, if it exists. Partial filenames may also designate directories, or devices. When a filename names a directory, all files in that directory are printed. For instance, to print out all files in the directory of frm on device */snd:*

% lsf /snd/frm

Equivalently, if your working directory is also */snd*, the command

% lsf /frm

works identically. On the other hand,

% lsf frm

assumes that frm is either a sound file or a subdirectory in your current sound file directory. The statement

% lsf /snd

prints out all files recursively on that device.

Flags

-l print long form. Gives name, read/write status, protection, file type *(Scratch, Hold,* or *Keep),* size of file in cylinders, whether it is contiguous or not, and the date. Which

date is printed depends on the **-tX** flag. Creation date is printed by default.

-f print the sound file descriptor. Information includes sampling rate, number of samples, channels, etc.

-R recursively search directories. Turned on automatically when a device is specified.

-pX print only files with substring X in filename. This is a poor-man's version of "regular expressions". Normal regular expressions don't work for **lsf** because the directory searching is done by the shell, which looks in the UNIX file system for obvious reasons, and knows nothing of the sound file system. This mechanism gives a primitive searching capability. The string X is taken as a key, and if the key is found in a filename, **lsf** prints it out. The substring is applied only to actual filenames, not directories.

-tX sort files according to date. If there is a character X appended, that will determine which date associated with the file will be used:
X = c sort by creation date (default)
X = a sort by date last accessed
X = m sort by date last modified

-C sort files according to size in cylinders.

-r reverse sort.

-a print all files. This prints out all files regardless of whether they are SDF files or not.

-0..-9 The flags, -0, -1, ... -9, provide the mechanism whereby one can manage a system file dump program. The algorithm is to return the names of the files which have changed since the last lower level dump. For information on how to manage a dump program, see the manual page for dumpsf(1csound).

AUTHOR

Gareth Loy

SEE ALSO

dumpsf(1csound) sndio(1csound), rmsf(1csound), mvsf(1csound), sfck(1csound).

NAME

m4n - read floatsams from M for N samples

SYNOPSIS

m4n [-RL] -bM -dN < floatsams > text or floatsams

flags:
-RN set sampling rate to N (16384)
-bM set begin time to M
-dN set duration to N
the usual complement of expressions and postoperators are allowed

DESCRIPTION

m4n reads floatsams on its standard input and copies them to its standard output, skipping until sample M, copying for N samples.

If the standard output is a terminal, arabic sample numbers are printed, otherwise floatsams are produced.

NAME

median – nonlinear filter for outputting time-varying median

SYNOPSIS

median [-wN] [-RN] < floatsams > floatsams

flags: (default)

-wN set window size to N seconds (default: 9S)

-RN set sample rate to N (read from stdin)

Arguments may be expressions. Use postop 'S' for samples.

DESCRIPTION

This program implements a median filter. The median filter is a nonlinear filter; its output at each point in time is the median of the N most recent input values. The attraction of such a filter is that - unlike a linear filter - it can remove glitches while still following step changes perfectly. On the other hand, the median filter does introduce a signal-dependent distortion which depends also on the window duration. The optimal size of the window depends on the duration of the glitches which are to be removed. In general, the window must be at least twice as long as the longest glitch which it is to remove. (The window is always centered about the "current" sample.) The primary application of this program is expected to be cleaning up time-varying pitch estimates.

NOTE: If the input to *median* is greater 1.0 in magnitude and the output of *median* is piped to *sndout*, then the -of flag must be given to *sndout* so that the stored values will be floating point as opposed to binary.

AUTHOR

Mark Dolson

NAME

mixsf – add, subtract, multiply, or interleave soundfiles

SYNOPSIS

mixsf [flags] soundfile < floatsams > floatsams

flags:
a: add soundfile to stdin
s: subtract soundfile from stdin
m: multiply soundfile by stdin
c: divdide stdin by (energy) soundfile (for AGC)
i: interleave soundfile with stdin
x = output scale factor to avoid clipping (1.)
g = input gain factor for soundfile (1.)
b = begin time in soundfile (first sample to use) (0.)
e = end time in soundfile (last sample to use) (end)
d = duration of soundfile (end - begin)
q = quiet time before soundfile (delay before first sample)(0)

DESCRIPTION

mixsf allows two soundfiles to be added (-a), subtracted (-s), multiplied (-m), divided (-c) (e.g., a signal is divided by its temporal envelope to implement a kind of automatic gain control - AGC) or interleaved (-i) (e.g., two mono files become separate channels of one stereo file). This can always be done using *cmusic*, so the attraction of *mixsf* is simply that it allows these operations to be performed much more easily. *mixsf* also provides a certain amount of amplitude and timing control.

The basic idea is that one soundfile is input via sndin on stdin while the other soundfile is specified by a filename following all other flags (e.g., see *sfnorm*). The output to stdout always has the same duration as the input to stdin; however, the soundfile specified in the commandline can be either shorter or longer than this. If it is shorter, then zeros are added to fill out the remaining time; if it is longer, then the excess is never used. The -b, -e, and/or -d flags can be used to specify that only a specific section of this soundfile should be used. The -q flag causes zeros to be added at the beginning of this soundfile.

The user is responsible for ensuring that arithmetic overflow does not occur. This is controlled by the -x flag. For example, if two soundfiles are being added, then -x.5 will guarantee that no clipping can occur. This is to be distinguished from the -g flag which is also a gain control, but which applies only to the input from the soundfile specified in the commandline.

AUTHOR

Mark Dolson

NAME

mixsnd - csound file mixing program

SYNOPSIS

mixsnd [flags] < score > samples

flags:

-v verbose, print a digested form of the score on stderr,

-bN make sound file buffer size = N (default 16K)

-s share open sound file descriptors if two notes play the same file

DESCRIPTION

mixsnd is a simple, monophonic, score-driven *csound* file mixer. It reads its standard input for text lines of the form:

note begin_time filename duration segment# offset gain;

where:

note is the literal string, "note",

begin_time is the time to start mixing in the file,

filename is a (possibly partial) *csound* filename specification,

duration is the length of time to mix in the file, durations less than 0 specify the end of the sound file,

segment# is the segment number, if the sound file has segmentation info, (currently unimplemented, this field is ignored)

offset is an offset into the sound file of where to begin,

gain is the gain.

It ignores all lines that do not begin with *note* or *not*, so you can include comments, *ter* statements and the like, which have no effect.

Example score:

```
note 0 /snd1/frm/av1 4 .37 -12dB;
note 1 twiddle 2 0 1/3;
note 1 /dgl/fiddle 4 24KS .25;
```

Expression parsing similar to **cmusic** is included via **expr()** in **libfrm.**

mixsnd writes the mixed samples on the standard output.

All files are opened as soon as they are encountered in the note list and remain open throughout the duration of the program. This is done to reduce the overhead of *csound* file opening and closing.

THE FLAGS

-v causes a digested version of the score to be printed on the standard error output to verify interpretation of the score. All time values are printed first in seconds, then samples, enclosed in "[]".

-bN causes the buffer size for each sound file to be set to N. By default it is 16K samples. If many files are open however, this can backfire if the program starts getting swapped very much. Shortening the buffers even to 1K produces acceptable throughput.

-s causes different notes that share the same sound file to use the same sound file descriptor. Again, this is an economy designed to share buffer memory where possible. It is not advised where two notes play from different parts of the same sound file at the same time, as this causes considerably greater disk i/o latency.

AUTHOR

Gareth Loy

SEE ALSO

sndout(1carl), sndin(1carl). For more robust file mixing, use the **sndfile** unit generator in **cmusic(1CARL)**.

NAME

mksfdir - make a sound file directory

SYNOPSIS

mksfdir directory1 directory2 ...

DESCRIPTION

mksfdir creates a directory in the sound file system. It is similar in function to the UNIX command **mkdir.** It can be used both to create subdirectories from your home sound file directory, or (if you are the superuser or the sound file superuser) to give new users home directories.

mksfdir automatically determines whether you are the superuser or sound file superuser before allowing a path outside your own area to be created. Note this restricts the creation of new home directories on csound filesystems to root or the disk pseudouser.

AUTHOR

Gareth Loy

SEE ALSO

sndio(1csound), mvsf(1csound).

DIAGNOSTICS

If you do not own the sound file path leading up to the directory you are trying to create, it will say, "no permission to create path." If the directory already exists, it will say, "directory already exists."

NAME

mm – convert .m4 Makefile prototypes

SYNOPSIS

mm

DESCRIPTION

mm takes the file *Makefile.m4* in the current directory and runs it through *m4(1)* redirecting its output so that the file *Makefile* is produced.

The contents of *Makefile.m4* are expected to be in the syntax of *make(1)*, and also expected to include a system directory path configuration file (see below under FILES). Macros found in this include file allow for system-independent configuration of Makefiles.

The *Makefile* has its mode set to readonly to remind you that the file *Makefile.m4* should be edited instead. **mm** removes any existing Makefile before compiling its replacement.

FILES

m4INCLUDE/config.m4 contains a database specifying site-specific directory paths for CARL software. m4SRCDIR/Makefile.c is a prototypical Makefile which can be taken and customized for most applications.

SEE ALSO

"Computer Audio Research Laboratory Software Writers Guide," CARL Technical Memorandum.

NAME

mountsf, umountsf - mount and unmount a csound volume

SYNOPSIS

mountsf csound_file_system
umountsf csound_file_system
umountsf up1

DESCRIPTION

mountsf and **umountsf** are Cshell scripts which manage mounting and unmounting media on which sound file systems have been built.

More information is available on how to mount/unmount packs at CARL by saying "help local/udp".

FILES

/etc/fstab, /carl/lib/sfstab (may be in /usr/local/lib instead)

NAME

mpg – keep track of gas mileage

SYNOPSIS

mpg [– v] [– t] [mileage_log]

DESCRIPTION

mpg reads a *mileage_log* (or standard input) and computes the gas mileage from it.

The format of the *mileage_log* is as follows:

<odometer> <gallons> <cost> <company>

The <company> field is optional and is used when the – v flag is given. Lines beginning with a sharp sign (#) are ignored. Anything after the fourth field is ignored. Here is a sample *mileage_log*:

```
# 1983 honda 650cc nighthawk
# odo   gal   cost   company   date
20098.2 1.935 2.65   mobil     27jul84
20215.8 2.527 3.46   mobil     30jul84
20317.8 2.455 3.41   mobil     25sep84
20415.8 2.083 3.04   mobil     01oct84
20502.0 2.2   3.13   texaco    04oct84
20583.0 2.1   2.83   texaco    08oct84
20681.4 1.967 2.87   mobil     11oct84
20782.5 2.5   3.28   texaco    15oct84
20907.5 2.4   3.24   texaco    18oct84
21024.2 2.624 3.96   mobil     23oct84
21143.3 2.4   3.42   texaco    28oct84
21235.2 1.967 2.87   mobil     03nov84
21343.0 2.28  3.27   texaco    09nov84
21458.9 2.234 3.26   mobil     14nov84
21562.1 2.350 3.43   mobil     17nov84
21663.5 2.236 2.86   mobil     20nov84   started using regular unleaded
21782.7 2.578 3.22   mobil     28nov84
```

The output of **mpg** is fairly self-explanatory. It consists of the total cumulative mileage covered by the *mileage_log* file, the total gallons, the total amount of money spent, the average mileage, and the average cost per 100 miles.

If the – t is given it causes a *trace* of the computations, for each odometer reading it gives the average mileage and average cost per 100 miles up to that point.

If the – v flag is given the output of **mpg** is slightly more verbose and the fourth field of the *mileage_log* file comes into play; for each gas company a total amount of money spent there, the highest cost of a gallon of gas there, the lowest cost of a gallon of gas there, the average cost of a gallon of gase there, and the number of visits there is displayed.

The cost per 100 miles figure is given in order to demonstrate the fallacy of the claim of the advertisers that their premium gas (high test, ethel, etc.) has more power and therefore gives better mileage. One might be misled into believing that if premium has more power it gives better mileage and this imporoved mileage offsets its extra cost and that there is a savings, but this is not the case.

The easiest way to keep track of your gas mileage is to let the tank get almost completely empty and then fill it up and write down the odometer reading. Thereafter each time you get more gas simply record the odometer reading and the amount of gas bought.

AUTHOR

Rusty Wright

NAME

mvsf - move sound file

SYNOPSIS

mvsf [-f] source destination

DESCRIPTION

mvsf moves the source sound file to the destination, similarly to the regular UNIX mv command.

AUTHOR

Gareth Loy

SEE ALSO

sndin(1csound), sndout(1csound).

DIAGNOSTICS

The move will be aborted if the destination file already exists. To override this, supply the -f (for "force") flag.

Moving a file to itself is resisted.

There are restrictions on file names. In particular, regular expression syntax will not work since the shell interprets these for the UNIX file system before it gets to **mvsf**. However, the notation "." for the destination file name is supported.

NAME

newwire – template program for inline digital sound signal processing

SYNOPSIS

newwire < floatsams > floatsams

DESCRIPTION

newwire reads floatsams from stdin, and writes them on stdout. Floatsams are binary, floating-point sample numbers. If the output is connected to a terminal, the floatsams are formatted to print on a screen. All by itself, it is a signal processing "no-op".

While **newwire** has a useful function in itself (described in its source code comments), it was written to serve as a model for other CARL signal processing programs. An informed user can take the source code for **newwire** (it is written in C) and transform it into a signal processing program. It uses several other programs developed at the Computer Audio Research Laboratory (CARL) at UCSD's Center for Music Experiment (CME).

FILES

getfloat.c, putfloat.c, expr.c, polish.c

AUTHOR

F. R. Moore

SEE ALSO

para(1carl), chan(1carl), impulse(1carl), getfloat(3carl), expr(3carl)

NAME

noise – generate various flavors of random noise

SYNOPSIS

noise [-h] [-l] [-u] [-n] [-S] [-w] [-s] [-r] [-m] > output

DESCRIPTION

flags: (default)

-h prints help message

-l lower bound (-1.0)

-u upper bound (+ 1.0)

-n number of samples (16384)

-S seed (0)

-w weighting (w), options are: w = white, f = 1/f, i = rational white b = brownian, m = mixed

-s sequence length (4096) for 1/f low sequence lengths tend towards whiter noise

-r relative distance traveled for brownian (.1) range: 0 (correlated) => 1 (uncorrelated)

-m mixture of noise type for -wm mode: -1 => 0 mix b and f, 0 => +1 mix f and w

noise either prints random numbers on your terminal or if the standard output is a file or pipe, it writes floatsams. By default, it produces 1 second of white noise at a sampling rate of 16KHz. The noise is bounded by default to the signed unit interval, [-1,+1]. Use the -l and -u flags to change this range.

1/f noise is implemented using Voss's algorithm. The sequence length must be a power of 2 (it is truncated to the nearest lower power if you miss). Low sequence lengths tend to whiten the noise. A sequence length of 1 is exactly white noise.

The *-r* flag, when Brownian noise is selected, determines the distance that the next point can be from the current one, expressed as a fraction of the total range between the upper and lower bound. Typical values: .25 approximates the spectrum for 1/f noise, .1 gives 10dB/octave rolloff, 1 gives whiteish noise.

The *rational* noise mode gives whole numbers only. These are necessarily outside the range of the signed unit interval. You must always use -l and -u to set an appropriate range for your application. Output is still floating point, but without fraction.

Finally, the spectra can be varied by mixing Brownian, 1/f and white noise in varying proportions. The -m flag acts like a panpot: between -1 and 0 it produces a correctly weighted mixture of Brownian and 1/f noise. At 0 it is exactly 1/f noise. Between 0 and +1 it goes from 1/f to white. For this mode, you may also vary the -s and -r flags to tailor things.

AUTHOR

Gareth Loy

NAME

nonzero - output non-zero floatsams

SYNOPSIS

nonzero [-RN] < floatsams > output

flag:

-RN set sample rate to N (default: 16384)

DESCRIPTION

nonzero reads floatsams on its standard input. If output is a file or a pipe, non-zero floatsams are written. If output is a terminal, sample #, time re. 16384 sampling rate, and value are given.

NAME

noteanal – note event parser for sound files

SYNOPSIS

noteanal [flags] < floatsams > floatsams,
Input must be a file or pipe.

Flags:
All time values are in seconds. Use 'S' postop for sample times.
-wN = simple average using window size of N (128S)
-mN = mean squared using window size of N (128S)
-bN = set segment begin threshold to N (.01)
-eN = set segment end threshold to N (.005); -eN must be <= -bN
-uN = set minimum segment duration (1024S)
-lF = set log file for segmentation statistics
-s = turn on segmentation (b, e, l and u automatically set this)
-uN = set min. interesting utterance size to N (1024S),
-x = skip output by window-size
-z = output the input sample rather than the average
-v = verbose: print summary of noteanal on stderr
-RN = print segmentation statistics using sampling rate N (48K).
-h = help. Prints this synopsis.

DESCRIPTION

noteanal reads 32 bit floating point sound sample data (floatsams) from the standard input. It produces two different outputs depending on what it is told to do. It's default mode is to simply be an envelope follower, writing on the standard output the mean squared envelope of the entire file. Simple averaging is also available with -w.

If prodded, it will instead try to segment the file by looking for amplitude divisions between events. The way it does this is to form a running average of so many contiguous samples of the file. The size of the window is set by the argument to the -m (and -w) flags. If this average goes above a settable beginning threshold (-b), a segment is started. It then begins looking for the average to drop below a minimum (-e), indicating the end of the segment. It repeats the process for the entire file.

The result of segmenting depends on the destination of the output. If the standard output is a file or pipe, the samples of each segment are written out as floatsams. If it is a terminal, the sample number (relative to the beginning of the file), time (calculated at the current sampling rate, settable by -R) and current average value are printed. Samples in between segments are suppressed.

It is possible to get a terse version of the envelope with the -x flag, which skips the output over the number of samples in the window. The number of output points is thus divided by the size of the window. This is good for previewing or quick analysis.

Segmentation is enabled explicitly with the -s flag, which will do segmentation using all defaults. The -b flag sets the beginning threshold, -e the ending. The -u flag determines the minimum length segment worth handling and is used to eliminate false starts and hiccups. Mentioning any of the -e, -b, -u or -l flags automatically turns on -s.

You can get **noteanal** to output the input samples instead of the envelope of the samples with the -z flag. This, in conjunction with -s and friends, effectively compresses inter-event silence from the sound file.

A summary of the segmentation activity can be obtained with the -l flag. It prints in the file named as its option, and provides the following info for each segment: segment number

beginning sample, (followed by beginning time in seconds, re. the current sampling rate, printed in parenthesis), the maximum average amplitude within the segment, the sample (and time) where this maximum occurs, and the sample (and time) of the end of the segment.

These segmentation statistics can be written to the standard error output with the -v flag.

Example

Imagine the file in this example contains speech.

% sndin speechfile |noteanal -u.1 -z -llogfile |sndout ...

Segmentation is turned on by the -u flag being mentioned. Segments less than .1 sec (at the default sampling rate) will be ignored. For most speech, the default value of -u works fine. The -z flag causes **noteanal** to write the actual samples read in, rather than writing the envelopes of the segments. The file "logfile" is written, containing the text description of the segments.

Tweaking The Parameters

In plain envelope mode, only -w affects anything. It changes the number of samples which are averaged. The greater the number, the smoother the envelope, and the greater the phase lag of the envelope behind the signal.

In segmentation mode, increasing -w makes a smoother average. The results are to slightly retard the onset of a segment (phase lag), and to allow only broader changes in the envelope to trigger a segment boundary.

Increasing -b makes it harder for a bump in the envelope to qualify as the beginning of a segment. Decreasing it makes it more sensitive. Increasing -e makes it easier to **end** a segment, since a higher threshold will presumably be reached quicker in a note's decay than a lower one. The -e value must be less than the -b value, or segments will end as soon as they begin.

Increasing -u causes a segment which has a duration less than the value set to be ignored. Decreasing it has little influence if it is decreased down below the size segment that is being found with -w, -b and -e.

EXPRESSIONS

A modified version of the CARL expr() routine evaluates all numeric arguments, so that expressions involving "+ - * / ( )", the postoperators 'K', 'S' and 's', are available. 'K' multiplies the result by 1024. For flags that set time values, 'S' treats the result of an expression as number of samples, while no postoperator, or the 's' postoperator treats time in seconds at the prevailing sampling rate.

AUTHOR

Gareth Loy

DIAGNOSTICS

A limit of 2048 segments maximum is enforced. The program prints "too many segments" and proceeds to overstrike the last segment recorded with the next and all subsequent segments.

A 0 or negative sampling rate quits with an error.

NAME

ochan – multiplexed sound signal processing with parallel pipes

SYNOPSIS

ochan #_of_parallel_channels process_1
... **process_N** > floatsams

DESCRIPTION

ochan provides multichannel signal processing by multiplexing its standard output, collecting each channel (through pipes) from one of several monophonic processing pipelines which run as parallel processes.

ochan runs parallel processes that take no input and multiplexes their outputs onto a single stream.

ochan 2 "wave" !...

takes the outputs of two copies of wave, and multiplexes it onto **ochan's** standard output.

AUTHOR

F. R. Moore

SEE ALSO

ochan(1carl), chan(1carl), para(1carl), newwire(1carl), getfloat(3carl)

NAME

offset - add constant offset to floatsam stream

SYNOPSIS

offset value < floatsams > floatsams

DESCRIPTION

offset reads floatsams (32-bit binary floating point samples) on its standard input and adds the constant value offset. The result is either printed on the standard output or written as floatsams depending on whether the output is connected to a terminal or a pipe.

AUTHOR

F. R. Moore

NAME

opensf - closesf - user-level commands to open/close sound files.

SYNOPSIS

opensf [flags] [file] > sound_file_descriptor

closesf files

or

closesf < sound_file_descriptor

flags for

opensf:

-w open file for writing
-r open file for reading
-rw open file for read/writing

Additional flags for write mode: (default in parenthesis)

-cN channels: 1, 2, 4, (default: 1)
-RN sampling rate: positive integers, (default: 49152)
-oc output packing s (short), f (float), (default: s)
-TN file size (expressed as time in seconds), (default: approx. 3s)
-CN file size (expressed in cylinders) (default: 1)
-tc realtime flag: r (realtime), n (not realtime), (default: n)
-rs remark: quoted text string
-Is include file: unix filename
-pN file protection: 0000 - 0666, (default: 0644)

N is a number, c is a character, s is a quoted string.

DESCRIPTION

Flags for **opensf** are a subset of those for **sndout(1csound)**. Omitted are the -i flag and -b, -e, and -d flags.

The function of **opensf** and **closesf** is to allow other than *csound* programs to access the *csound* system in a controlled way. Ordinarily, a program's only access to *csound* files is via **sndin** and **sndout**, which read/write from/to the standard input/output, respectively. However, it is useful for non-csound system programs to be able to access sound files. For instance, **cmusic**, which is not a program in the *csound* system, can still read *csound* files via its *sndfile* unit generator.

csound system uses the UNIX file protection scheme to assure the integrity of *csound* system tables and files. At CARL, all files, directories and raw devices associated with the *csound* system are given a protection code which allows write access only to a pseudo-user named *disk*, and all *csound* programs are made set-user-id to this pseudo-user. Thus, all *csound* i/o is done under the name of this pseudo-user. But this mechanism does not make it easy for others to write programs that use the *csound* file management routines. The problem is then how to let user programs other than those owned by *disk* access *csound* files?

The mechanism for updating a *csound* file from an unprivileged process is for that process to execute an **opensf**() or **closesf**() subroutine calls. These subroutines invoke the **opensf** and **closesf** program via **fork**(). Since the programs **opensf** and **closesf** have disk pseudo-user permission, then can initiate the open or close request.

In a typical sequence, the user process invokes the subroutine **opensf()** which forks an **opensf** process and requests a read operation. **opensf** returns the sound file descriptor *(sfd)* for the file on its standard output which is read in by **opensf()** in the user process. The user process then uses this *sfd* to access the disk with **sndi()** or **fsndi()**. The user process then calls **closesf()** which forks a **closesf** process to close the sound file.

FILE MODES AND ALLOCATION

New files are created by default as *non-realtime*. This simply means that the system does not need to fit the file into one contiguous block. No space is claimed for a new file so created until the first attempt to put something in it. A new file will be made *realtime* by mentioning any of the following flags:

-tn sets file to realtime and claims 1 cylinder by default;

-TN sets file to realtime and claims enough cylinders for time in seconds requested.

-CN sets file to realtime and N cylinders.

SEE ALSO

sndin(1csound), sndout(1csound), opensf(3csound), closesf(3csound).

NAME

para – parallel sound signal processing with common input & summed output

SYNOPSIS

para < input > output

DESCRIPTION

para provides parallel signal processing by feeding its stdin (through pipes) to each of several monophonic processing pipelines which run in parallel, summing the outputs of these processes, and writing the resulting summed data stream on the standard output. The *input* and *output* data streams normally consist of binary, floating-point sample numbers. If the output is connected to a terminal, ASCII data is produced. A typical command form:

```
cmusic score.sc |para 2 "comb .6 1000" "comb .7 1131" |sndout
```

applies both "comb .6 1000" and "comb .7 1131" to separate copies of the output of cmusic, then adds the parallel signals and pipes the sum to sndout. If only one process is specified for multiple channels, it is applied in parallel to each channel.

If only one process is specified, it is applied in each parallel process. The number of processes must be 2, 3 or 4. The parallel processes must be enclosed in quote (") marks, and may themselves be pipes, such as

```
cmusic score.sc |para 4 "comb .6 2131 |comb .6 1000" |sndout
```

AUTHOR

F. R. Moore

SEE ALSO

newwire(1carl), chan(1carl), getfloat(3carl)

NAME

peak – calculates peak amplitude of sound signal

SYNOPSIS

peak [**flags**] < floatsams > peak (ASCII display)
or
peak [**flags**] – e < floatsams > floatsams, and > & peak
flags:

-error	copy stdin to stdout, print peak on stderr
-dB	print peak amplitude as dB amplitude
-absolute	(default) detect absolute maximum
-negative	detect negative maximum
-positive	detect positive maximum

Input must be a file or pipe of floatsams.

DESCRIPTION

peak reads samples from stdin and finds the maximum amplitude. The input data stream must be floatsams. In the first usage given above, it prints the index in the file of the peak, and the peak value, on stdout. In the second usage, when the -e flag is given, the peak is written to stderr, and the input samples are copied to stdout. This allows peak to be inserted in a sample data stream as a "test probe."

The optional flag -dB (abbreviating this to -d will work too) causes the amplitude value to be printed in dB.

The peak is ordinarily detected from the absolute value of the signal. Use the -positive flag to detect only positive signals, and – negative for negative signals. These can be abbreviated -p and -n.

The following examples determine the peak amplitude for sound file boom:

```
sndin boom |peak
sndin boom |peak -e |sndout boom2 (peak placed on stderr)
```

AUTHOR

F. R. Moore

SEE ALSO

rms(1carl), hist(1carl)

NAME

pianoroll – CARL score analysis program

SYNOPSIS

pianoroll [-g] [-m] [-M] [-x] [-y] [-d] [-b] [-e] [-s] [-h] < input.sc > output

DESCRIPTION

flags: (default)

-g y axis grain size (.1)

-m minimum value on x axis (0)

-M maximum value on x axis (1000)

-x P field to display on x axis (6)

-y P field to display on y axis (2)

-d P field to determine duration (4)

-b display score starting at time (0)

-e display score until time (infinity)

-s also show sum of x values

-h displays this summary

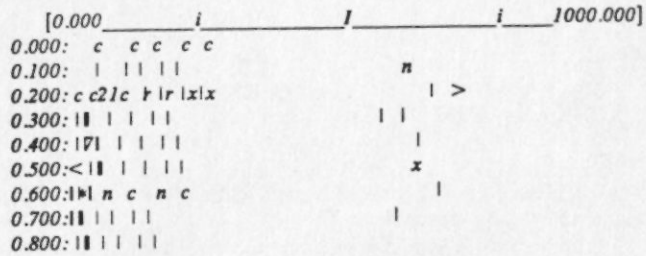
pianoroll reads a standard *cmusic* score file, and creates a graphic display of the score on its standard output in pseudo-pianoroll notation. The graphic format utilizes standard teletype characters. By default, it displays time (P2 and P4) along the y axis (as does a piano roll) and frequency (or whatever is in P6) along the x axis. Duration of a note (P4) is displayed by the vertical length of the line.

The Flags

The *g* flag sets the "grain" size (0.1), which is the y axis quantization. This ordinarily means how much time passes for each line of the pianoroll drawn. The *m* and *M* flags determine the minimum (0) and maximum (1000) values respectively that will be displayed on the x axis. The *x* and *y* flags set which P fields (P6 and P2 respectively) to sample for the x and y axies. The *d* flag specifies the P field used to determine the duration of the note. *b* and *e* determine the begin and end points of the score to be displayed. The *s* flag is used to display the sum of the values of the P field being shown in the x axis. This is useful e.g., to view amplitude fluctuations through time. See below.

The Display

Here is an example display. It is a bit illegible because it demonstrates several features of the display for pathological cases.



The beginning of a note is represented by the first character of its name, at the appropriate location in the x axis. Thus, at time 0, a chord using an instrument whose first character is 'c'

is played. At time .1, the chord is continued, and voice 'n' enters near 1000Hz. At time .2, several things happen. Instruments 'r', 'c', and 'x' enter. The numerals '2' and '1' indicate that more than one instrument were active within the character position for that time/frequency quantum. What is represented is the number of collisions. If the number exceeds '9', e.g., at time .6, a '*' (meaning "many") is printed. At time .2 and .5, the frequency of the notes being played went off scale, and this is represented by '<' and '>'. Note that the duration of the off-scale event(s) is still recorded. If it is desired to disambiguate the '*'s and '<'s, use a smaller grain size and/or an expanded x axis range.

Score Format

The score must have the value to be displayed in the y axis (P2) sorted in monotonically increasing order. All statements besides those beginning with the three characters "not" are ignored. ("note" is also accepted.)

P fields may contain constant arithmetic expressions governed by the same rules applied to *cmusic* expressions. However, **pianoroll** has no knowledge of variables, strings, macros, or P field substitution. These must all be evaluated prior to handing the score to pianoroll. The most convenient way of doing this is to run the score through *cmusic* with the statement

```
set barefile = filename;
```

at the top. A copy of the score with all expressions, macros and variables expanded and substituted will be produced in the named file. (*cmusic* still produces all the sample data this way. If you just want the barefile, try running with a reduced sampling rate, do-nothing instruments, and the like.)

Alternate Techniques

pianoroll will allow any P fields to be substituted for P2, P4 and P6. This can be useful for viewing non-standard scores, or for plotting any parameters against each other.

One very useful approach is to show amplitude vs. time. If amplitude is represented by P5, you would use flag -x5. Since amplitudes are typically between 0 and 1, you would rescale the maximum x range with -M1. What one is usually after here is to see the sum of the amplitudes to detect where it might be overflowing. The -s flag turns on a feature that adds the display of the sum of the P fields active during that quantum. Plus signs, '+' are used to show the sum.

Here is a score, showing amplitude through time.

```
[0.000_____i_____I_____i_____1.000]
0.000: 1cc  c     +
0.100: 311  1      +
0.200: 733  3            +
0.300: 1 1                  +
0.400: 1 1                  +
0.500: 1 1                  +
0.600: 1cc  c     +
0.700: 1 1     +
0.800: 1 1     +
0.900: 1cc  c     +
1.000: 1 1     +
```

As we can see, most amplitudes in this example are below 0.1. The maximum sum is about .45 at time 0.3, where there are 16 voices active.

More exotic possibilities are obtainable this way. For instance, how about a plot of frequency vs. amplitude?

AUTHOR

Gareth Loy

SEE ALSO

cmusic(1carl)

DIAGNOSTICS

Notes out of monotonic order are reported and skipped. Requesting display of a P field that does not appear is reported and skipped. If a P field cannot be numerically evaluated, it is skipped.

BUGS

Remembering that when instruments collide in a quantum, the NUMBER of instruments in that quantum is recorded rather than the first character of a name; it becomes clear that it is not a good idea to name your instruments beginning with numbers.

A limit of 128 P fields can be handled.

Where two values collide in a quantum, if -s is turned on, the colliding values are kept track of as one. The amplitude calculated is still correct, but it may not remain correct. If one of the colliding notes ends sooner than the other, its amplitude is not subtracted until all notes occupying that quantum have ended. Thus, false peaks may appear in very dense sections. However, expanding the x axis, and/or decreasing the grain size until the collision is avoided will reveal the correct amplitude.

NAME

pitch – program for estimating time-varying pitch

SYNOPSIS

pitch

DESCRIPTION

pitch [flags] < floatsams > floatsams

flags: (defaults in parenthesis)

-RN input sample rate set to N (usually read from stdin)
-rN output sample rate set to N (default is 256 samples/sec)
-lN lower pitch boundary set to N (50)
-uN upper pitch boundary set to N (200)

This program expects a single-channel, single-pitch input stream such as from a recording of a voice or of a solo instrument. It's output is a time-varying estimate of the pitch at an output sample rate which is typically 256 estimates per second. The algorithm is the parallel processing scheme of Rabiner (see *Digital Processing of Speech Signals*, Rabiner & Schafer, pp. 135-141, 1978). Six simple peak detectors operate in parallel, and a decision matrix is used to determine the current best estimate of the pitch.

Because this algorithm is very fast and simple, it is also very prone to error. Three things can be done to help it:

1) Filter the input file to eliminate frequencies below 50 Hz and above 1000 Hz (if appropriate). This can be done by using *fastfir* to design a bandpass filter (typically with -n511) and then using *convolve* to do the filtering.

2) Use the -l and -u flags to specify the lower and upper limits on the estimated pitch.

3) Use a post-processor such as *median* to smooth the pitch estimates and eliminate gross errors.

FILES

AUTHOR

Mark Dolson

SEE ALSO

BUGS

NAME

play – play sound file(s) through DACs

SYNOPSIS

play [flags1] [filename1] [flags2] [filename2] ... [flagsN] [filenameN]

flags:

-p prompts you before starting to play file

-i interactive play mode (see below)

-qN inserts N secs. silence before file

-rN repeats file N times (no gap between iterations)

-bN start file at time N

-eN end file at time N

-RN force file to be N sampling rate

-FN force use of filter N (0, 1, 2, 3) (default tracks the sampling rate)

-Dw,x,y,z...
set dacs to use; can be any of {1, 2, 3, 4}

-v describe in detail what play is doing

-g make flags not global

Defaults: tries to play file test in your current sound file directory if no name is given. The right anti-aliasing filter is selected automatically if no -F is given. *Nota Bene:* Once a value is set by a flag, it continues in effect across subsequent files unless -g is given, in which case flags only affect the file they appear before.

DESCRIPTION

play sends the named sound files to the DACs. It opens them all first so the inter-file setup time is minimized.

Pressing the [DEL] key aborts immediately.

The flag structure is different for this program than any other sound file program in that flags only affect the subsequently named sound file (with rare exceptions such as -i, see below). If both a silence flag and repeat flag precede a file, the silence only applies to the period before the first time the file is played. All repetitions are without intervening silences. (Repeats with silence can be obtained by nameing the file more than once with -q in between.) Example:

play -p -q10 -r3 buzz -q3 toot -r2 foo bar

Here, you will be prompted after 10 seconds of silence, and before playing buzz. Buzz will be repeated 3 times without intervening silence. After 3 seconds of silence, toot will play once, followed immediately by two times through foo, and once through bar. Inter-file silences are constrained by UNIX to be integer multiples of one second.

Various default actions of **play** can be overridden. To play through a channel besides 1, use the -D flag, e.g., -D4 uses channel 4. For a stereo file, try -D1,3. Override the sampling rate with -R, e.g., -R32K.

The -g flag causes all flags to loose their global interpretation for the rest of the files to be set up. Thus,

% play -D4 foo baz

plays both foo and baz through dac channel 4, but

% play -g -D4 foo baz

plays baz through the default channel 1. Flags copied are: -q, -r, -q, -F, -D, -b, -e. Note -R is not copied, thus files of different sampling rate preserve their own rates.

Interactive Mode

Mentioning the -i flag starts up play's interactive mode. All files named are then treated interactively. After opening all the named files, a status line and a '*' prompt are displayed. The status line shows the current file, its current begin time, end time, sampling rate, repeat count, silence count, and converters selected. Most actions of **play** can be modified from this point, according to the following rules. (In these rules, objects inside "[]" are optional. Rules followed by '*' can be repeated any number of times).

x	exit play
[e,b,d]=N	set [end,begin,duration] time to N
N	change last altered parameter to N
[e,b,d][>,<][N]	move [e,b,d] [ahead,back] by N or if N is missing, by increment
!	play the file
!!	repeat the last command
n	and go on to next file
p	go back to previous file
r=N	repeat N times
[>,<]*[N]	move [ahead,back] begin/end play times by * increments, increment modified by N
i=N	set increment to N
R=N	set sampling rate to N
D=S	set DAC channels to string S for mono, S is a single number, for multi-channel, it is a comma-separated list of numbers
q=N	set N seconds of silence preceding play of the file
I	enter fast interactive mode
v=1	set verbose mode, v=0 resets it
f=[m,s,ms,S]	set print format of begin/end times on status line to minutes, seconds, milliseconds, samples.
F=[0,1,2,3]	set lowpass filters, 0=12.8kHz, 1=6.4kHz, 2=bypass, 3=non-ex

Time values such as begin and end default to seconds. Expressions are allowed for all instances of N. Expressions may contain postoperators as follows:

k	times 1000
K	times 1024
S	time in samples
s	time in seconds (default)
ms	time in milliseconds
m	time in minutes

Expressions setting the value of a time variable (b,e,d) may be the character '$' which stands for the file boundary time. For the expression "e=$", e is set to the end of file, for "b=$, b is set to 0, for "d=$", both b and e are set to the file boundary times. Caution must be exercised with postoperators in expressions, since the postoperators commute across the entire expression. For example, the following attempt to set b to one sample before one second,

```
* b=1-1S
```

will be interpreted instead as

```
* b=1S-1S
```

or 0. The effective way to say this is:

```
* b=1-(1S)
```

Only one command may appear per line , except '!' which may appear separately or at the end of any other command. If any text follows the single '!' command, it is assumed to be a UNIX command. In this case, the window is not played. Instead, the samples in the window are piped to the named command. This allows the following kind of command:

```
* !sndout segment
```

which captures the current window in a sound file named segment. Caution must be exercised with this command, it does not behave like a similar command in the vi(1) or ex(1) text editors. In particular, the samples in the window are always written, so you couldn't do a command like, e.g.,

```
* !date
```

to find the time of day, since the date command knows nothing about getting samples on its stdin! Also, beware of this:

```
* !sndout segment &
```

Fast Interactive Mode

From interactive mode, you can enter "fast interactive mode", which has a slightly different syntax that allows immediate play on each keystroke, for making very fast adjustments to a window. In the following list, those commands that immediately play the window are indicated by "(*)". All the rest do not immediately play.

Fast Interactive Mode Commands:

x	to quit fast interactive mode,
!	plays current window (*),
b or e	subsequent cmds move only begin or end time,
> or <	move forewards/backwards by increment (*),
w	subsequent commands move window,
b=N or e=N	set begin or end time to N,
i=N	set increment to N

Segment Editing

In addition to the interactive commands, the following set of commands allows one to create, read, alter and/or write information about segments of a file.

play has several constructs that expedite editing segments of sound files. They are

the window

which is the difference of the time of the e and b variables.

the named segment list buffers

Named segment list buffers must be UPPER CASE characters A - Z. A segment list buffer can be used to keep around a list of segments. Thus, you can identify segments on one list, save it in a named buffer, and make other lists, modify values on lists, etc.

the unnamed list buffer

which is used for temporary lists.

In the following, N and M are either expressions, or the character '$'; S is a segment specification of the form "s1", "s2", etc.; L is a named list buffer as in sA. The '$' when appearing in the context of a segment number stands for the last segment, and when appearing as a time stands for the end of the current file.

Segment Editing Commands

sr [file]	read segment file,
sw [file][=L]	write segment file, without [=L], writes out all segments, with [=L], writes only segments on list L.

sp[L]	print segments, without [L], prints all segments. with [L], prints only segments on list L.
sN	set begin/end playing times to segment N.
sb[N]=[M,S]	set begin time of segment N to number M or segment S, default for N = current segment, default for M = current begin/end times, S is another segment spec. like se5.
sb=N or se=N	set current begin/end time to time N.
sbX=N or seX=N	set begin/end time of segment X in unnamed list buffer to time N.
sbX=sbY or sbX=seY	set begin/end time of segment X to begin/end time of segment Y.
s=	set begin/end times of current segment in unnamed list buffer to playing times.
sN=	set begin/end times of segment N to playing times
sm	make new segment using begin/end playing times, add to unnamed list buffer.
sd[N]	delete segment N, if no N, current segment, from unnamed list buffer.
s=N,N,...N!	make temporary segment list from segment number list N and play it.
sL=N,N,...N	store list of segment numbers N,...N in named list buffer L.
sL=M	copy list buffer M into list buffer L.
sL=	copy unnamed list buffer into named list buffer L.
sL	fetch segment list from named list buffer L into unnamed buffer.
st[L]	time the creation of segments by hitting [CTRL]-\ optionally store in list buffer L.

Timing Mode

In normal interactive mode (not fast interactive) there is a way to play a file (or section of it) and hit a key whenever you want to mark a segment boundary on the file. This is called timeing mode.

To do this, issue the command

```
* st
```

which says to set up the file's default segment list buffer to automatically save segments that are created every time you hit a certain key. You then start the file playing by giving the usual command:

```
* !
```

(the command can be given at one fell swoop, of course, as "st!"). While the file is playing, you hit the [CTRL]-\ keys whenever you want to mark the end of a segment. The current time (measured in seconds from the time in the file where you started playing) is printed at that point. You can use any of the labeled segment list buffers (A - Z, uppercase only) to save the automatically created segments by the command:

```
* stA
```

for instance, which will use segment list buffer A instead of the file's default segment list buffer.

If the things you want to mark just go by too fast for you to get, set the sampling rate to a lower number. The times of the segments will all be adjusted by the ratio of current_rate/actual_rate, so the segments will still have the correct times at the actual_rate speed.

This operation also succeeds on playing just a short segment of the file. Set the begin and end times to the desired values, and proceed as usual. The begin time reported for the first segment is the time at which playing began.

AUTHOR

Gareth Loy

SEE ALSO

record(1x), sndio(1x). For tutorial information about interactive mode, say

help csound/play

DIAGNOSTICS

If any file is not found, or an illegal flag is encountered, the process aborts. Any DAC errors are reported with a quasi-English error message, plus the value of the DAC DMA status register and the converter's ASC status register. Command errors in interactive mode induces a help message.

BUGS

The syntax for interactive play is very strange. While it is very efficient (requires very few keystrokes), it is consequently rather arcane. It is also difficult to extend. It should have been done with lex and yacc.

The mechanism for silences is to call sleep(), which only gives time quantization to the one second level. Thus, the period of silence can be off by as much as .5 secs in either direction.

There is a limit of 64 files that can be played at one time.

NAME

polish – convert mathematical expression to reverse polish notation

SYNOPSIS

```
#include <carl/frm.h>

char *polish(expression, unops, binops, postops)
char *expression, *unops, *binops, *postops;

cc [ -flags ] files -lfrm
```

DESCRIPTION

polish processes 4 strings as arguments: *expression* is a string containing an expression to be transformed into reverse polish notation, and *unops*, *binops*, and *postops* are strings describing unary, binary, and post operators to be recognized in the expression. Anything in the expression not recognized as an operator is taken to be an operand. A pointer to a (static) string containing the polish form of *expression* is returned.

The operator strings are ordered lists of sets, with operators in the first set done before operators in the second set, which are done before operators in the third, etc. Sets are notated as a list of comma-separated items enclosed in braces ("{}"). For example, the string "{*,/}{+,-}" as *binops* would specify that "*" and "/" are to be done before "+" and "-".

Unary operators are always done before binary, and binary before post operators.

Operators may be more than single characters; in particular, functions and post operators are may be treated as multicharacter *unops* and *postops*.

Polish returns a string in reverse polish notation consisting of comma-separated fields, each containing a symbolic item (either an operator or an operand) followed by a dollar sign, followed by a digit. The digit is the number of operands of the item, which is zero for operands, 1 for unary and post operators, and 2 for binary operators. (Someday function calls, which are treated as UNOPS, may allow more than 1 argument as well).

Example:

```
polish("-3+ (-p4-ln(v3)*p5)Hz",
  "{sin,cos,ln}{-}","{*,/}{+,-}","{dB,Hz}");
```

will return the following string:

```
"3$0,-$1,p4$0,-$1,v3$0,ln$1,p5$0,*$2,-$2,+ $2,Hz$1,"
```

AUTHOR

F. R. Moore

SEE ALSO

expr(3carl)

NAME

getprop, putprop addprop rmprop printprop putheader getheader cpioheader cpoheader noautocp stdheader getpaddr getplist putplist – routines to read, write and edit headers on sound sample streams

SYNOPSIS

```
#include <carl/carl.h>
#include <carl/procom.h>
#include <carl/defaults.h>

char * getprop(iop, name)
    FILE *iop; char *name;

putprop(iop, name, value)
    FILE *iop; char *name, *value;

addprop(iop, name, value)
    FILE *iop; char *name, *value;

rmprop(iop, n)
    FILE *iop; char *n;

printprop(proplist, outp)
    PROP *proplist; FILE *outp;

putheader(iop)
    FILE *iop;

PROP * getheader(iop)
    FILE *iop;

cpioheader(ip, op)
    FILE *ip; FILE *op;

cpoheader(pl, op)
    PROP *pl; FILE *op;

noautocp()

stdheader(iop, name, srate, nchans, format)
    FILE *iop; char *name, *srate, *nchans,

PROP * getpaddr(iop, name)
    FILE *iop; char *name;

PROP * getplist(iop) FILE *iop;

putplist(prop, iop)
    PROP *prop; FILE *iop;
```

DESCRIPTION

getprop(iop, name)

Return property *value* associated with *name* from property list associated with FILE pointer *iop*.

putprop(iop, name, value)
Make a property with the *name/value* pair, and put it at the absolute end of the property list associated with the FILE pointer *iop*. This does absolute concatenation; for wizards only, use *addprop for normal property list insertion*.

addprop(iop, name, value)
Make a property with the *name value* pair, and insert it just before the TAIL of the property list associated with the FILE pointer *iop*.

rmprop(iop, n)
Remove property with name *n* from property list on *iop*.

printprop(proplist, outp)
Print property list pointed to by *proplist* on FILE pointer *outp*.

putheader(iop)
Write out property list built on FILE pointer *iop*.

PROP * getheader(iop)
Return the address of the head of the property list for FILE pointer *iop*.

cpioheader(ip, op)
Copy property list on *ip* to *op*.

cpoheader(pl, op)
Copy property list *pl* to FILE pointer *op*. Note: this does not write the property list. A subsequent call to **putheader()** is required.

noautocp()
Suppress automatic header copy of *stdin* to *stdout*. Ordinarily, when **putfloat()** is called after **getfloat()**, the header read with **getfloat()** is automatically copied and written on *stdout* before the samples are written. This suppresses the copy.

stdheader(iop, name, srate, nchans, format)
Make a standard header with the *name, sampling rate, number of channels* and *sample format* in it, and make it be the property list for FILE pointer *iop*.

PROP * getpaddr(iop, name)
Return the address of the property element matching the *name* on FILE *iop*.

PROP * getplist(iop)
Return the address of the head of the property list for *iop*. It is equivalent to saying **getpaddr(iop, H_HEAD)**.

putplist(prop, iop)
Make property list *prop* be the property list of FILE pointer *iop*.

EXEGESIS

The normal course of events is for a program to want to read a header prior to doing data input, and to examine what it finds there for particular properties. **getheader()**, reads up the header. It can be called either before or after the reading of input data, (**getfloat()** and friends automatically read and skip over any undigested header) but **getheader()** is typically done first. Calls to **getprop()** follow to examine the header for specific properties. (Note: a call to **getheader()** must always preceed the first call to **getprop()** for a particular *iop.)* Then, typically, a program wants to construct an output header with **putprop()**. After it is built, it can be explicitly written out with **putheader()**, or the first call to an output routine such as **putfloat()** will write it automatically.

Ordinarily, when a program reads the *stdin*, and then writes to the *stdout*, any header on the *stdin* is automatically copied to the *stdout*. In this way, programs that do nothing with headers, and even are innocent of the whole notion of headers, but which still use the standard versions

of *getfloat()* and friends described in getfloat(3carl) will at least transparently pass the header through untouched.

addprop() does not delete any properties that may have the same name before adding a property. You must do this with **rmprop()** if you are trying to replace a property on a list.

FILE

/carl/lib/libcarl.a

AUTHOR

Gareth Loy

SEE ALSO

wire(1carl), floatsam(5carl), getfloat(3carl). Also, "Procom - Interprocess Sample Data Communications Facility for UNIX," by Gareth Loy, CARL Technical Report.

NAME

purgesf - lists csound files which are ripe to be deleted

SYNOPSIS

purgesf [flags] directory

flags:

-S lists scratch files that can be deleted
-H lists hold files that can be deleted
if neither -S nor -H is given, list both
-l long form listing
-D sorts by referenced date, oldest first
-tN lists files unreferenced for more than N hours
-n only prints number of cylinders reclaimable
-U lists users affected by purge
-N uses "notify" format
-dN day N (N = = # days in future) included in purge note with -N

DESCRIPTION

purgesf nominates files to be removed so as to free up working space on a sound file system. It can be used in conjunction with **reapsf(1carl)** and **dumpsf(1carl)** to manage the culling and cold storing of the nominated files.

purgesf lists files that have been unreferenced for more than certain thresholds of time. Default time thresholds are set by the *csound* file system manager. Other time thresholds can be set via flags. **purgesf** merely reports lists of files that satisfy certain criteria, it does not actually delete anything, and has no side effects.

If either the **-S** or **-H** flags are used, only files in that category that have exceed their reference threshold will be listed. The threshold can be modified with the **-tN** flag where **N** is is the number of hours a file must have been unreferenced in order to be listed. If neither **-S** or **-H** flags are supplied, **purgesf** first lists all *Scratch* files, then all **Hold** files that exceed their reference thresholds. (In this case, the **-t** flag is ignored.)

The **-S** and **-H** flags cause two actions each:

-S the age threshold is adjusted to 24 hours, and only files with *Scratch* mode classification are nominated. (If a **-tN** flag has also been given, its threshold overrides that of the default.)

-H the age threshold is adjusted to 96 hours (4 days), and only files with *Hold* mode classification are nominated. (Again, a **-tN** specification overrides the default, only when one or the other is given.)

Ordinarily, the files are sorted by user. They can be sorted by date, longest-unreferenced-file first, by including the **-D** flag.

A "long form" listing of the files may be obtained by adding the **-l** flag. This produces the filename, size in cylinders, the kind of file (S = = Scratch, H = = Hold), and its last referenced date. This also generates a line at the bottom of the listing indicating the total reclaimable cylinders.

The **-n** flag suppresses all but the listing of the total reclaimable cylinders.

The **-u** flag lists the names of the users whose files have been nominated by **purgesf.**

The **-N** flag causes the output to be suitable to be mailed to the users whose files have been named. The message header is

The following sound files have been unreferenced since: (date specified by threshold) *and are therefor subject to purging.*

Then all the nominated files are listed.

The -dN flag takes N as the number of days in the future when a purge will be done. When used in conjunction with -N, the message header is augmented with the lines:

The purge is scheduled for: (date of purge)

AUTHOR

Gareth Loy

SEE ALSO

sndin(1csound), sndout(1csound).

NAME

getfloat, fgetfloat, putfloat, fputfloat, flushfloat, fflushfloat, getshort, fgetshort, putshort, fputshort, flushshort, fflushshort, fgetfbuf, fputfbuf, fgetsbuf, fputsbuf – read and write sound sample streams from UNIX files or pipes

SYNOPSIS

```
#include <stdio.h> #include <carl/carl.h>

int getfloat(floatptr)
float *floatptr;

int fgetfloat(floatptr, iop)
float *floatptr;
FILE *iop;

int putfloat(floatptr)
float *floatptr;

int fputfloat(floatptr, iop)
float *floatptr;
FILE *iop;

int flushfloat()

int fflushfloat(iop)
FILE *iop;

int getshort(shortptr)
short *shortptr;

int fgetshort(shortptr, iop)
short *shortptr;
FILE *iop;

int putshort(shortptr)
short *shortptr;

int fputshort(shortptr, iop)
short *shortptr;
FILE *iop;

int flushshort()

int fflushshort(iop)
FILE *iop;

fgetfbuf(fp, n, iop)
float *fp;
short n;
FILE *iop;

fputfbuf(fp, n, iop)
float *fp;
```

```
short n;
FILE *iop;

fgetsbuf(sp, n, iop)
short *fp;
short n;
FILE *iop;

fputsbuf(sp, n, iop)
short *fp;
short n;
FILE *iop;
```

DESCRIPTION

These routines form the core of conventional data i/o routines used by all CARL programs. They work in a way analogous to getchar(3) and putchar(3). They return a positive value if successful, 0 on EOF (all the *get* routines), and a negative value on error.

Sample data is transmitted between programs as binary floating point samples called *floatsams*. *Shortsams* are binary short integers. The term *floatsam* is used generically on occasion to mean both kinds. Most programs only deal with floatsams, except those programs that specifically say they read both, or that convert from one format to another.

THE ROUTINES

getfloat() reads a single floatsam from the standard input. Note, that the address of the float must be passed to **getfloat()**. **getshort()** reads a single shortsam from the standard input. These are both actually macros for the following functions where iop is defined as stdin.

fgetfloat() and **fgetshort()** read from the file opened on iop.

putfloat() and **putshort()** take the address of a float/short-sam to write to stdout. They are macros for **fputfloat()** and **fputshort()** with iop defined as stdout.

flushfloat() and **flushshort()** **MUST** be called before exiting programs that have used **putfloat()** or **putshort()** in order that the last buffer of data be written on the file. These are macros for **fflushfloat()** and **fflushshort()** with iop defined as stdout.

fgetfbuf() and **fgetsbuf()** read buffered blocks of floatsams and shortsams respectively. **fputfbuf()** and **fputsbuf()** write buffered blocks of floatsams and shortsams respectively.

HEADERS

All these routines are able to detect the presence of headers on the data as defined in headers(5carl). The headers are stripped off, and set aside where the header routines described in headers(3carl) can find them.

FILE

(at CARL) /usr/local/lib/libcarl.a.

AUTHOR

F. R. Moore conceived getfloat() and putfloat(). Gareth Loy implemented the full isomorphic set, and added headers.

SEE ALSO

newwire(1carl), floatsam(5carl), procom(3carl).

NAME

pvoc - phase vocoder

SYNOPSIS

pvoc [**-flags**] < floatsams > floatsams

DESCRIPTION

pvoc is the program which actually implements the phase vocoder. The phase vocoder is a signal processing technique which can be used to produce very high fidelity modifications of an arbitrary input sound. It can be used as an analysis-synthesis system to independently modify duration and pitch, or it can be used simply to perform analysis, with the data being subsequently utilized in cmusic or elsewhere. The one drawback to the phase vocoder is that it requires considerable amounts of computation time and of soundfile disk space; however, intelligent selection of parameter values can do much to alleviate this problem.

The phase vocoder can be viewed as a bank of bandpass filters, but with one additional complication: whereas the output of a normal bandpass filter is simply a bandpass-filtered version of its input, the outputs of the phase vocoder bandpass filters are the time-varying amplitude and frequency of the bandpass-filtered signal. For example, if the input signal is a tone of well defined pitch, and if the phase vocoder bandpass filters are set up so that exactly one harmonic of the input signal lies within each filter bandpass, then the outputs of the phase vocoder will be the instantaneous amplitudes and frequencies of each harmonic.

Actually, this only describes the analysis part of the phase vocoder; much of the attraction of the phase vocoder is that it can also recombine the analysis data to produce a perfect resynthesis of the original input. In addition, it can modify the analysis data to produce resynthesis of arbitrary duration without altering the pitch. In fact, very high fidelity time-scale modification can often be obtained even when the pitch is unknown (or not well defined) or even when the input is polyphonic. The key requirement is simply that the phase vocoder have enough filters (of narrow enough bandwidth) so that the entire spectrum of the input is covered, but never with more than a single harmonic in any one filter bandpass.

The phase vocoder recognizes the following flags:

-R input sampling rate. The default is the input sampling rate listed in the header of stdin; hence, this flag is RARELY USED. If the header value is not correct (or if there is no header) the -R flag MUST be used to specify the correct value; failure to do this will render the analysis data useless.

-N number of bandpass filters. The default is 256. Filters will be centered at 0 Hz, R/N Hz, 2R/N Hz, ..., (N-1)R/N Hz. Typically, N should be chosen so that R/N is less than the lowest pitch in the input sound; this ensures that no more than one harmonic will ever fall within a given filter (assuming that the filter bandwidth is not greater than the separation between adjacent filters). For polyphonic sounds, more filters may be required. The phase vocoder runs most efficiently when N is a highly composite number (e.g., a power of two), but any even value will be accepted. (Odd values of N will be internally rounded up.)

-F fundamental frequency. This is an alternative to specifying N directly; DON'T use both -N and -F. The phase vocoder simply sets N = R/F or F = R/N depending on which is specified. (A specified value of F may be slightly readjusted internally to make N even.)

-M length of filter impulse response. The default is N-1. NOTE: It is far more common to specify -W than -M (see below). The filter half-bandwidth (i.e., width from center of passband to edge of stopband) will be given approximately by 2R/M (for a hamming filter); hence, a longer impulse response gives a proportionally narrower bandwidth. For good analysis results, M should be on the order of 4*N-1; for good

time scaling, it is better to let M = N-1 (or even N/2-1). (But sometimes 4•N-1 works better for time-scaling too!) Any odd value will be accepted. (Even values of M will be internally rounded down).

-W filter bandwidth factor. This is an alternative to specifying M directly; DON'T use both -W and -M. The phase vocoder simply sets M = 4•N-1, 2•N-1, N-1, or N/2-1 corresponding to -W0, -W1, -W2, or -W3. So the above suggestions for M translate to usually using W0 for analysis and W3 (or W0) for synthesis. Intermediate values (e.g., W1 and W2) are less frequently used.

-D decimation factor. The default will be automatically calculated as D = M/8 (D = M/(8T) if time expansion by a factor of T>1 is specified). The default is the maximum recommended decimation factor. This flag should NOT be specified in normal use.

-I interpolation factor. The default is I = D (I = T•D if time scaling by a factor of T is specified). This flag is ALMOST NEVER used.

-L length of synthesis filter impulse response. The default is L = M. This flag is ALMOST NEVER used.

-T time scale expansion factor. The default is T = 1. Integer values of T give the best results for speech; for music, non-integers work equally well. Large values will not necessarily sound good! NOTE: The time-scale factor is readjusted internally. The actual expansion (or compression factor) will be the ratio of the integers I/D.

-P pitch-transposition factor. The default is P = 1. The actual transposition factor will be a ratio of integers N/NO which will hopefully be equal to I/D. This is most likely to work if N is a large number; but, in fact, pitch transposition is far less reliable than time-scaling. (One solution to this is to implement all pitch transposition in two distinct steps: 1) time-scale by desired pitch factor (e.g., 1.0595 for a half-step), 2) sample-rate convert by inverse of this factor (e.g., input_rate/output_rate = 1./1.0595). But note that even this can only be done reliably by specifying I and D directly instead of T.)

-w warp spectral envelope. The default is w = 1. This is an experimental feature which can be used to preserve the spectral envelope while performing pitch-transposition. For example, if P = 2., then specifying -w with no argument will produce appropriate warping for P = 2 (i.e., -w2). A value of -w different from -P can be specified if desired.

-K flag for using Kaiser filter instead of hamming filter. This flag is ALMOST NEVER used.

-A flag for analysis only. The results should be piped to sndout with a file name ending in ".i" (or something similar) to remind the user that this sound file contains analysis data and should NOT be played. The format is N+2 channels (N/2 + 1 amplitude channels and N/2 + 1 frequency channels all interleaved), with odd channels being amplitudes and even channels being frequencies (channels are numbered 1 thru N+2). The first two channels contain the amplitude and frequency of the DC (zero frequency) filter, so the data for the fundamental is usually in channels 3 (amplitude) and 4 (frequency).

-E flag for analysis only with spectral envelope output instead of amplitude and frequency. The format is single channel at a sample rate of (N/2 + 1) * R / D. (i.e., the spectral envelope is calculated every R/D input samples; N/2 + 1 values are required to specify the spectral envelope.)

-S flag for synthesis only. The input is assumed to have been created by a previous

run of the phase vocoder using all of the same flags (except -A in place of -S). DON'T specify both -A and -S.

-V flag for creating summary text file. The default name for this file is pvoc.s; this can be changed by entering the desired new file name as the last item on the command line (after all of the flags). This flag can be useful for keeping accurate records of phase vocoder runs. However, with large numbers of channels, it may be wise to edit it down to some smaller size using vi.

A typical use of the phase vocoder for analysis might be specified as:

sndin in_file |pvoc -F440 -W0 -A -V stat_file |sndout out_file

A typical use of the phase vocoder for time scaling might be specified as:

sndin in_file |pvoc -N1024 -W3 -T4 |sndout out_file

A detailed description of usage (and suggestions for optimal usage) are contained in the helpfile for this program. However, one point which is well worth stressing here is that the input soundfile should NEVER be at a high sampling rate unless the run in question is the absolutely final version of a production run! Failure to observe this rule will slow the computer down horribly and pointlessly.

DIAGNOSTICS

Diagnostics originating in pvoc are only concerned with catching bad flag specifications and are intended to be self explanatory. Probably the most cryptic is "warning P= something not equal to T= something_else"; this occurs when pitch transposition by some unobtainable factor is attempted. If the listed P and T differ by more than .1 or so, it may help to try again with a larger N.

BUGS

Sound inputs with peak amplitudes near 1. can sometimes result in modified versions with peak amplitudes greater than 1.; this will result in a particularly nasty form of overflow when piped to sndout. The solution is to rescale the input using "gain".

Although the phase vocoder works well for a surprisingly wide variety of sound transformations, there are some cases where it simply will not perform acceptably. These are areas for further research!

AUTHOR

Mark Dolson

FILES

NAME

pwsf - print working sound file directory

SYNOPSIS

pwsf

DESCRIPTION

pwsf will print your current *csound* file system directory path. It is similar in meaning and usage to the UNIX command **pwd.**

SEE ALSO

cdsf(1csound), lsf(1csound).

NAME

quad – sound path interpreter for cmusic

SYNOPSIS

quad [flags] **path_file** [seconds_duration]

DESCRIPTION

flags:

-LN	length of function quad returns
-a	produce global amplitude function
-r	produce global reverberation function
-d[N]	produce doppler shift function (optional format, see text)
-1	produce channel 1 function
-2	produce channel 2 function
-3	produce channel 3 function
-4	produce channel 4 function
-vN	set the velocity of sound in m/sec. to N (doppler shift only)
-oN	set the degrees offset from x axis for loc. of channel 1 to N
-t	calculate individual channel amplitude linearly (see text)

quad is a *gen* function for *cmusic(1carl)*. It reads a *path_file* containing sound trajectory information consisting of [x,y] pairs representing the succession of points a sound is to occupy through time. **quad** interprets this path and produces a set of functions that implement the location modulation scheme described in [J. Chowning, "The Simulation of Moving Sound Sources", **JAES**, January, 1971, Vol. 19, #1.]

The first argument to **quad** specifies the number of points in the cmusic function it is to generate (typically 1024). The next argument specifies which of the seven possible functions **quad** will produce on this run of the program. This is followed by the file containing the path, as produced by *sndpath*.

The syntax for specifying doppler shift is slightly different since you must also specify the duration of the sound to calculate the correct frequency shift function. In this case, you can either supply the duration as a number concatenated to the -d flag, as in -d5, or provide the value as a separate argument after the pathname. The -v flag provides a way to adjust the velocity of sound, which defaults to 340 m/s.

The -o flag takes a number of degrees offset to use in calculating the location of the speakers. Ordinarily, channel 1 is defined as 45 degrees above the x axis in quadrant 1. The locations of the other channels are computed from this. For instance, -o90 calculates the function for channel 1 as though it were positioned directly in front of the audience, with the other speakers forming a diamond figure around the audience.

The -t flag causes the equations

$$channel[N] = \sqrt{(1.0 - coeff)/(\pi/2)}$$
$$channel[N+1] = \sqrt{coeff/(\pi/2)}$$

to be used in computing the amplitude functions for the individual channels. By default, the equations

$$channel[N] = \sqrt{1.0 - .5*(1.0 + \tan(coeff - (\pi/2)/2.0))}$$
$$channel[N+1] = \sqrt{0.5*(1.0 + \tan(coeff - (\pi/2)/2.0))}$$

are used.

EXAMPLES

As with all *cmusic* gen programs, **quad** can be run standalone.

Example shell calls:
% quad -L1024 -a pathfile
% quad -L1024 -d5 pathfile
% quad -L1024 -d pathfile 5
produces 1024 floatsams (binary 32 bit floating point samples) on the standard output. Note that commands 2 and 3 are equivalent. The output can be reviewed with e.g., show(1carl).

An example call from a **cmusic** score might be:

```
var s1 "-a ";
var s2 "pathfile";
gen 0 quad f1 s1 s2;
```

Note the blank after the -a in string variable **s1**. Otherwise the flag would be run together with the pathfile when passed to **quad.**

Here is a complete example *cmusic* score. This example does no reverberation, which is left to the interested hacker as an exercise.

```
#define DUR 10
set quad;

ins 0 quadrille;
        seg b1 1 f3 d 0; /* doppler shift */
        mult b1 b1 p6;          /* multiplied against base freq */
        osc b2 p5 p8 f2 d;      /* amp. envelope */
        osc b1 b2 b1 f1 d;      /* carrier */

        seg b2 1 f4 d 0; /* global amp */
        mult b1 b2 b1;          /* in b1 */

        seg b2 1 f5 d 0; /* chan 1 */
        mult b3 b2 b1;          /* in b3 */
        seg b2 1 f6 d 0; /* chan 2 */
        mult b4 b2 b1;          /* in b4 */
        seg b2 1 f7 d 0; /* chan 3 */
        mult b5 b2 b1;          /* in b5 */
        seg b2 1 f8 d 0; /* chan 4 */
        mult b6 b2 b1;          /* in b6 */

        out b3 b4 b5 b6;
end;

gen 0 gen2 f1 1 1;                      /* sine wave */
gen 0 gen1 f2  0 0  .1 1  .6 .1 1 0;    /* amp envelope for note */
var 0 s1 "-d loop.s";                   /* doppler function lasts DUR seconds */
gen 0 quad f3 s1 DUR;
var 0 s1 "-a loop.s";                   /* global amp. function */
gen 0 quad f4 s1;
var 0 s1 "-1 loop.s";                   /* channel 1 */
gen 0 quad f5 s1;
var 0 s1 "-2 loop.s";                   /* channel 2 */
```

```
gen 0 quad f6 s1;
var 0 s1 "-3 loop.s";                    /* channel 3 */
gen 0 quad f7 s1;
var 0 s1 "-4 loop.s";                    /* channel 4 */
gen 0 quad f8 s1;

note 0 quadrille DUR 1 1000Hz p4sec .06sec;
ter;
```

AUTHOR

Gareth Loy.

Considerations

All *cmusic* gen functions must accept -c and -o flags for "closed" and "open" mode. **quad** only usefully produces -c format functions, so these flags are no-ops.

When the sound path travels within the radius of 1 meter of the listener's head, the global amplitude function (which would otherwise exceed 1.0) is truncated to 1.0.

SEE ALSO

sndpath(1carl), cmusic(1carl) John Chowning, "The Simulation of Moving Sound Sources", *JAES*, January, 1971, Vol. 19, #1.

NAME

r18 - read Stanford COPY format 18-bit sound sample tapes

SYNOPSIS

r18 > floatsams

DESCRIPTION

r18 reads Stanford COPY format 18-bit sound sample tapes and converts them to floatsams. It expects there to be a 128 word header which it will attempt to strip off unless a -H flag is given.

NAME

read_func_file, read_func_fid – routines to read floatsam sample streams into standard CARL FUNCTION format structures

SYNOPSIS

```
#include <carl/carl.h>
#include <carl/defaults.h>

FUNCTION * read_func_file(name, def_seq_type)
    char *name;
    char *def_seq_type;

FUNCTION * read_func_fid(fd, def_seq_type)
    FILE *fd;
    char *def_seq_type;
```

DESCRIPTION

read_func_fid()
: uses a FILE pointer to a file opened with fopen(3) to read a stream of *floatsams* with *fgetfloat(3carl)* and build an internal FUNCTION structure, described below.

read_func_file()
: is a front end for **read_func_fid()** which takes a file name to be opened.

EXEGESIS

These routines provide a uniform way of reading data which represent control functions (or arbitrary data) in floatsam format from a UNIX file into a program. **read_func_fid()** reads any header on the data with *getheader(3carl)* and uses the properties to determine the format of the FUNCTION structure. If the header contains a H_SEQUENCE property name with the property value of H_MONO_IN_X, the data is presumed to be an ordinary stream of floatsam values without abcissas. If the H_SEQUENCE property name has a value of H_XY_PAIRS, the data is presumed to be interleaved [x,y] floatsam pairs, representing piecewise linear functions. If the data has no header, the **def_seq_type** argument to the functions determines the default sequence type for the data. It should have a value of either H_MONO_IN_X, or H_XY_PAIRS. If a header is guaranteed, it can be NULL.

FUNCTION STRUCTURE

The FUNCTION declaration looks like this:

```
struct func {
        char *ftype;    /* function type: MONO_IN_X, or XY_PAIRS */
        char *fname;    /* function name */
        long flen;      /* function length */
        float *fxval;   /* x function values */
        float *fyval;   /* y function values */
};
typedef struct func FUNCTION;
```

The valid sequence types for the **ftype** subfield are H_MONO_IN_X for data monotonic in **x**, and H_XY_PAIRS for data in [x,y] pairs. For H_MONO_IN_X, the **fxval** pointer is set to NULL.

FILE

/carl/lib/libcarl.a

AUTHOR

Gareth Loy

SEE ALSO

genxy(1carl), floatsam(5carl), getfloat(3carl).

CODE EXAMPLE

A demo program which simply reads in functions and prints their contents is in /mnt/carl/src/demo/read_function.c at CARL.

NAME

readheader – read a header on a floatsam stream

SYNOPSIS

readheader < floatsams > header

or

readheader -e < floatsams > floatsams, and >& header

DESCRIPTION

In the first usage, **readheader** reads samples from the *stdin*, and prints any header it finds on the *stdout* in human-readable form. It throws away any input samples.

If the -e flag is given, **readheader** writes the input floatsams to *stdout* (the header will remain on the data) and the human-readable header is written to *stderr*. Thus, with the -e flag, **readheader** can be inserted as a "test probe" in a cascade of pipes to print out the header at that point, and to pass along the floatsams and the header untouched.

DIAGNOSTICS

If no header is present on the data, it says "readheader: no header".

AUTHOR

Gareth Loy

SEE ALSO

stripheader(1carl)

NAME

reapsf - reclaim temporarily used sound file space

SYNOPSIS

reapsf [flags] filesystem

flags:

-CN delete Scratch files first, then Hold files, as needed to reclaim N cylinders (default flag)

-SN only delete Scratch files as needed to reclaim N cylinders

-HN only delete Hold files as needed to reclaim N cylinders (reserved)

-y answer yes to all questions (reserved)

-t test, do not delete files

DESCRIPTION

reapsf reclaims temporarily used file space from the named sound file system.

There are two classes of reclaimable file space: Scratch and Hold sound files. A third class, Keep files, are unreclaimable by **reapsf.**

The normal operation is to use the – C flag to specify a quantity of space to reclaim. **reapsf will first try to delete** If it does not find enough that way, it will ask you if you want to reap Hold files for the remainder. If you answer anything other than "y" or "yes", only Scratch files will be removed. If you specify no flags, the default is to invoke – C, and specify the size of the sound file system as the number of cylinders to reclaim, which frees the maximum space on the disk that is available.

It is possible to select only one class for reaping with the – S and – H flags. Specify to **reapsf** the class of files to be removed and the quantity of space in cylinders to be freed. For instance, the flag **-S10** specifies 10 cylinders to be reclaimed by deleting Scratch files. The flag **-H10** does the same to Hold files. (Note: – H is a reserved flag, only wizards may delete Hold files without first deleting Scratch files.)

Implementation

The method used by the – C flag to **reapsf** is to make a list of files in the Scratch category, beginning with oldest Scratch files first, that have exceeded the Scratch file threshold. The list is terminated when the total cylinders exceeds the amount requested. If the amount requested is not achieved by removing all candidate Scratch files, you are asked if you wish to reap Hold files for the remainder. If you say yes, the same procedure is invoked for Hold files until the requested amount is reached or all candidate Hold files are tallied. The combined list of files is then deleted.

Age is determined by the last time the file **was referenced** by anyone. The reference time is the last time an operation, such as *sndin(1csound)* or *sndout(1csound)* was performed on a file. For *sndout* in particular, the reference time is the time at which *sndout* closes the file, not when it is created.

At CARL, Scratch files unreferenced for more than 24 hours, and Hold files unreferenced for more than four days, are reclaimable.

Usage

When you need more space for your work than is currently available, the method to get more is to first examine your own file usage. Remove any files that you know will not be of further use to you. If you are unable to delete enough files, then put some on magtape, or (if you have such an option) on another disk. Only when you have tried to solve your own problem should you run **reapsf.** Use the – C flag to remove scratch files first. Then, if that does not supply enough, remove hold files.

Note

It is possible, and even encouraged, for anyone to run reapsf at any time, on Scratch as well as Hold files, requesting as much space as they legitimately need to do their work. **reapsf** will *not* delete files that have not exceeded the aging thresholds.

BUGS

reapsf is a feeble attempt to enforce by administrative means that which can only be solved by a network of social interaction between users. There are two principal abuses of this program: either by maliciously protecting your files from being subject to **reapsf** or by maliciously reaping other's files when you know they are valuable and not backed up. If ever there were a program that depended on the Golden Rule, this is it.

AUTHOR

Gareth Loy

SEE ALSO

scratchsf(1csound), holdsf(1csound), keepsf(1csound), purgesf(1csound), rmsf(1csound), dumpsf(1csound), restorsf(1csound).

NAME

record - record sound file through analog-to-digital converters

SYNOPSIS

record [flags [file]]

flags: flag = (values:default)
-cN = set number of channels to N (1, 2, 4:1),
-RN = sample rate (49152),
-FN = set filters to N (0,1,2,3:0),
-q = quiet mode,
-TN = set file size to N (expressed as time in seconds),
-CN = set file size to N cylinders (1),
-r = remark(NULL),
-I = include(NULL),
-p = protection(0644)
-M = monitor mode
-B = broadcast mode
-h = print help message
File test in your current sound file directory is used by default. Filters used track the sampling rate unless specified with -F.

DESCRIPTION

record reads sound samples from the ADCs into the named *csound* file.

When run, it prompts you by saying:

press [RETURN] to record, anything else to abort.

and waits until you do so before immediately starting to record.

If the **delete** key is struck during a recording, the recording is stopped, and the length of the file truncated to the elapsed time. A message appears on the screen indicating the time at which the file will be truncated.

If the *core* signal is given by pressing **ctrl-** then the recording is aborted, and the program allows you to retry by jumping back to the prompt:

press [RETURN] to record, anything else to abort.

The flags behave in the same way they do in **sndout(1csound)**. The process is similar to **sndout**, since the only difference is the source of the samples. Differences from **sndout:** there is no -t flag (recorded files will all be realtime), and no -i or -o flags (packing is determined by the ADCs). Additional flags: -FN selects the numbered filter. At CARL, correct Nyquist filters exist for 49152 and 16384 sampling rates, and are selected by 0 and 1 respectively. **record** will set this automatically to the right one for the selected sampling rate. Filter 2 is BYPASS mode, i.e., no analog filtering of the output at all. Filter 3 is not installed at CARL. The -q flag is for quiet mode, no prompts, no diagnostics.

The flags, -M and -B set *monitor* mode and *broadcast* mode respectively. *Monitor* mode causes the digitized signal from an ADC to be wrapped immediately around to its like-numbered DAC. *Broadcast* mode causes the digitized signal from an ADC to be wrapped immediately around to all DACs. Whether this works on any given DSC system depends on the hardware configuration of the Nyquist filters. Refer to the Field Print set for your system.

SEE ALSO

play(1csound), sndio(1csound).

DIAGNOSTICS

While other recording rates than 49152 Hz are available (via the -R flag), these are quantized to the nearest available actual sampling rate supported by the DSC converters. You are notified of the actual sampling rate only when there is a difference. Appropriate filters are silently selected

automatically from the actual sampling rate.

Various errors may be reported, of which "disk error" and "converter error" are the two most frequent. The former indicates that, a buffer being sent to disk has not completed in time. The latter usually indicates that the converters have experienced a "data late" condition, indicating that they couldn't get enough throughput on the unibus.

AUTHOR

Gareth Loy

BUGS

File truncation via pressing the delete key seems to cause the length of the file to be slightly overestimated, by a value in the range of 8 to 80 milliseconds.

The converter system can be subject to occasional hiccups, resulting in clicks in your sound file for a variety of reasons. The usual reason is having too much conflicting activity on the disk or the unibus to support the required data rate. The only sure cure for this is to eliminate all vestiges of timesharing, running the system in single user mode. Or, if your data can suffer it, running at a slower sampling rate.

BANDWIDTH

Experience with our system has shown that with a single unibus VAX-11/780 system, where the disks and converters are on the same unibus (the worst case), an aggregate peak throughput is around 92160 samples per second, allowing for guaranteed mono recording up to 50ks. For the same VAX with either two unibuses or with massbus disks, the aggregate peak is such as to afford guaranteed stereo at 50ks. These are very conservative figures. Performance varies with the phase of the moon, sample rates may be slightly higher west of the Rockies, New York residents add 5% disk latency, don't call us, we'll call you.

NAME

rect - rectify floatsam stream

SYNOPSIS

rect < floatsams > floatsams

DESCRIPTION

rect reads floatsams on its standard input and writes the rectified floatsams on its standard output.

NAME

resamp - crude sample-rate conversion by linear interpolation

SYNOPSIS

resamp **-fFactor** < floatsams > floatsams

DESCRIPTION

resamp is a crude sample-rate conversion program which uses linear interpolation to do a quick-and-dirty sample-rate conversion by an arbitrary factor. Its main application is for factors (equivalent to l / m in srconv) which are between .9 and 1.1. These ratios may be difficult to obtain with small integers, so srconv may not be easily applicable in these cases. For example,

```
sndin file1 |resamp -f.93 |sndout file2
```

is equivalent to

```
sndin file1 |srconv 93 100 % |sndout file2
```

(i.e., it will slightly decrease the sample rate).

BUGS

At present, resamp simply writes the header from stdin out to stdout. This means that visf may have to be used to correctly list the new sampling rate.

AUTHOR

Mark Dolson

FILES

NAME

restorsf - restore sound files from magtape

SYNOPSIS

restorsf [flags] [-sSource -dDestination] files ...
or
restorsf [flags] < *file*
where *file* contains 1 or 2 sound file names per line. If one name per line, the named file is restored; if two, the first name is restored as the second name.

Flags:
-D print the complete tape directory
-L just list the file names in tape directory
-s Source file
-d Destination file
-V Verify that the tape can be read by restorsf
N.B., you must supply a list of files to be verified.
-v Verbose
-y Answer yes in advance to request to overwrite file.
-n Change mode of "contiguous" files on tape to be non-contiguous when restored.

DESCRIPTION

restorsf reads a magtape written by dumpsf(1csound) and copies the named files from the tape back to a sound file system.

Naming Names

There are three ways to name sound files to be restored.

File Input:

Saying
% restorsf < file
reads the contents of *file* which it expects to be in one of two formats. If *file* contains one filename per line, the named files are extracted from the tape and are restored with that name. If two names appear on one line, the first name is used to find the file on the tape, and the second specifies the name to restore it as.

Command Line

Files can be simply named on the command line. E.g.,
% restorsf fileX
restores *fileX* from the tape.

Alternately, source and destination names for a file can be given as – s and – d pairs. The file named after -s is the name of a source file on the tape, and it must be immediately followed by a -d flag and the name that you want to restore the file into. Note: there is no space between the -s and -d flags and the postpended filenames. Normal *csound* filename expansion is used. For example, if dgl is running this program and his current sound file directory is /snd/dgl,
% restorsf -s/snd/frm/tone -dfoo
restores /snd/frm/tone to /snd/dgl/foo

A useful way to generate a list of all files on a tape is with the command:
% restorsf -L > file
which writes a list of all files on the *dumpsf* series to *file*. An example of restoring all files from a dumpsf series is:
% restorsf -L > filename % restorsf < filename

The -D flag prints the dump directory.

The -V flag causes restorsf to just read the tape down to check it for readability. It does not restore any files. You must either supply restorsf with all the filenames on the tape, or with the name of the last file on the tape to get it to read.

The -n flag causes the mode of all files being restored to be set to "non-contiguous" mode. This comes in handy when there is not enough room in a single block on the disk to bring back a file from tape that was originally a contiguous file.

AUTHOR

Gareth Loy

SEE ALSO

dumpsf(1csound).

DIAGNOSTICS

If a file already exists on the sound file system and you own the file, you are asked if you want to overwrite it. If you do not own the path for where you want to restore the file, it will not be restored. A message indicating all requested files not restored for any reason is printed before **restorsf** exits. Note that a single run of *dumpsf* can generate multiple tapes. **restorsf** manages multi-file requests on multi-tape dumps, rewinding tapes and telling you what tapes to mount. If there is no room on the disk, you will get an error message and **restorsf** will continue with the next file (which might be to a different disk).

The -L flag lists all files on the series. When actually restoring them, **restorsf** will prompt you to mount the proper tape. Note, **restorsf** does not have to begin on the first tape of a multi-tape sequence, but can pick up with any tape.

A comment is included in the restored sound file indicating the source from which it was restored.

If **restorsf** is requested to restore a file that already exists on the disk, it will ask you if you really want to overwrite the file. The -y flag will automatically answer yes to this question.

NAME

retrosf - put retrograde of a sound file on standard output

SYNOPSIS

retrosf sound_file_name > floatsams

DESCRIPTION

retrosf produces the retrograde of a sound file on the standard output. It takes no flags.

NAME

reverb - one tap comb reverb

SYNOPSIS

reverb gain delay < floatsams > floatsams

DESCRIPTION

reverb reads floatsams (32-bit binary floating point samples) on its standard input and implements a simple comb filter on the signal with specified **gain** and **delay**. When the input signal is exhausted, it continues recirculating the output until the energy dies down to -60dB.

AUTHOR

F. R. Moore

NAME

rms – calculates rms amplitude of sound signal

SYNOPSIS

rms [-dB] < floatsams > rms (ASCII display)
or
rms [-dB] – e < floatsams > floatsams, and >& rms

DESCRIPTION

rms reads samples from stdin and finds the root mean square of the energy of the signal. The input data stream must be floatsams. In the first usage given above, it prints the rms value on stdout. In the second usage, when the -e flag is given, the rms is written to stderr, and the input samples are copied to stdout. This allows rms to be inserted in a sample data stream as a "test probe."

The optional flag -dB (abbreviating this to -d will work too) causes the rms value to be printed in dB. •

The following examples determine the rms amplitude for sound file boom:

```
sndin boom |rms
sndin boom |rms -e |sndout boom2 (rms placed on stderr)
```

AUTHOR

F. R. Moore

SEE ALSO

peak(1carl), energy(1carl)

NAME

rmsf - remove sound file(s)

SYNOPSIS

rmsf [-r] [-t] file ...

DESCRIPTION

rmsf removes the named sound files.

The -r flag specifies recursive deletion of the named sound file directories and all their subdirectories. You can test the action of -r before doing it for real with the -t flag, which simply reports what files would delete without doing so.

AUTHOR

Gareth Loy

SEE ALSO

sndio(1csound), mvsf(1csound).

DIAGNOSTICS

Prints "file not found" if a file can't be found, returns an error code equal to the number of unfound files. Various other errors indicating a scrambled sound file system are intercepted here.

NAME

sched – find common hours for meeting times

SYNOPSIS

sched [-p[X]] [[[day] [from]] [to]] < schedules > merged_schedules

DESCRIPTION

flag:

-p print individual schedules, If X is supplied, print schedule for participant named X

Without optional *day, from* and *to* arguments, **sched** prints a calendar. Optional *day, from* and *to* arguments indicate a sub-group of interesting hours, and cause a list of participants who can attend during those hours to be printed. Note: *day, from* and *to* arguments are not flags.

sched reads in text descriptions of schedules. It analyzes them for times in common, and prints a list of the hours when everyone is free.

The schedule format which **sched** reads from its standard input is as follows:

< name>
[[< day_of_week>] [from_hour to_hour] ...]
< name>
[[< day_of_week>] [from_hour to_hour] ...]

where < *name*> is the name of the person whose schedule follows. (Note: names must consist of only one word. See BUGS below). < *day_of_week*> is at least three characters specifying the day (the first letter may be upper or lower case, the rest must be lower case). The day is followed by any number of < *from_hour*> < *to_hour*> pairs which supply a range of hours for which that person is available on that day. The input format must specify hours in 24-hour time format. Any number of < *day_of_week*> lines may follow a < *name*> . It is assumed that for days not mentioned for a person the person is not available that whole day. The schedules of all participants to be compared are supplied one after the other.

Ordinarily, the output of **sched** is a 24-hour, seven day calendar. The calendar has asterisks ('*') for times when all parties can meet. Otherwise, the number of participants who can attend at a given hour is printed. If the number is zero, a vertical bar is printed (simply to help read the calendar).

The *p* flag causes **sched** to print the calendars it compiles for the individual participants. If the *p* flag has a name of one of the participants concatenated to it, just the calendar for that person is printed.

If any of the optional *day, from* and *to* arguments are supplied on the command line, **sched** takes this as a sub-group of time to examine. These values are in the same format as for the schedules. The names of the participants who can meet during that time interval are listed. Also listed is an indication of how many of the hours within that duration they can attend. If just a *day* is given, the whole day is examined. If just a *from* is added to this, the day from that time until midnight is examined.

USAGE

First compile a schedule file. To examine **sched** 's analysis of the schedules, use the *-p* flag to list them all. Name a person after the flag (e.g., *-pgareth)* to just see that person's schedule. Omit all arguments to get a meeting calendar. To examine a range of hours to see who can come during a particular time slot, give the *day, from* and *to* times in question. To examine an entire day this way, just mention the day.

AUTHOR

Gareth Loy

DIAGNOSTICS

The message "day expected on line X" indicates that either the day on line X was misspelled, or that a <from_hour> specification appeared without a corresponding <to_hour> on the previous line. The message "range format error on line X", indicates that a <to_hour> specification on line X was greater than or equal to its corresponding <from_hour> specification. This can occur, for instance, when hours are mistakenly specified in 12-hour format. The message "too many participants!" indicates that you have tried to schedule more than 512 participants, the current limit.

BUGS

Names of participants are restricted to one word. If you must use last names too, run them together with a non-blank character, such as hyphen or underbar. Hours must be specified in 24-hour format. Quantization is to the hour, so it is of no use for really tightly scheduled people. If more than 99 people can attend during one time slot, the calendar will become misaligned.

NAME

scratchsf - holdsf - keepsf - set volatility status of files.

SYNOPSIS

scratchsf filename
holdsf filename
keepsf filename

RATIONALE

On the UNIX file system, once a file is created it can only be deleted by its author, unless special permission has been granted by the author to the author's group or to others, or by the superuser. While this is suitable for general purpose file systems, it has special liabilities in such cases as the sound file system where a limited data storage area must be equitably shared, and where much of the data can be recreated at will, making absolute permanence much less desirable.

To this end, the sound file system implements a system which attempts to maximize the availability of free storage space for temporary use, and makes it harder to use the file system for permanent storage of sound. The sound file system supports three classes of volatility, designated *Keep*, *Hold*, and *Scratch*. *Scratch* files have a guaranteed lifetime of one day. *Hold* files have a guaranteed lifetime of seven (7) days. *Keep* files are not deleted, but they still may be dumped to magtape and purged if circumstances warrant. Depending on the weight of space usage, *Scratch* and *Hold* files may last considerably longer than the minimum time. It is anticipated that *Hold* files will be fairly long lived.

New sound files always start out life as *Scratch* files, and must be promoted to *Hold* or *Keep* status. Once you audition a file and decide you like it, you then put it on *Hold* status. *Keep* status is reserved for finished pieces, some demos, and other files that have received official blessing to have such status (Holy water is sprinkled and incense is burned...) The rule-of-thumb is: the harder it would be to recreate a sound file, the less volatile should be its keep status.

The mechanism for reclaiming space from *Scratch* and *Hold* files is to delete the oldest *Scratch* files first. Then, when all the *Scratch* files that can be deleted are gone, the same procedure is done on *Hold* files. Age of a file is determined by the last time the file was referenced. Thus, *Scratch* files may only be reclaimed one day after the time they were last referenced. *Hold* files will last 7 days after their last reference.

DESCRIPTION

When a file is created by **record, cpsf, catsf** or **sndout,** its class is set to *scratch*. For example, the sequence:

```
% wave |sndout test
% holdsf test
```

will preserve test for a minimum of 7 days after the last time it is referenced. Any number of files can be named to each command.

The copy of a file created with **cpsf** is set to *Scratch* mode. Renaming a file with **mvsf** keeps its former classification. Likewise, restoring a file from magtape via **restorsf** carries over its old classification.

To observe the classification of a file, use the

```
% lsf -l file
```

command, and look under the field marked "sc"; the mode will be either 'S', 'H' or 'K'. Or use the

```
% lsf -f file
```

command and look below the field marked *HOLD*. It will be '*S*' for *Scratch*, '*H*' for *Hold* and '*K*' for *Keep*.

RECLAIMING FILE SPACE

File space is reclaimed by the program **reapsf(1csound)**. See the manual page for that program.

NOTE: It is possible, and even encouraged, for anyone to run reapsf at any time, on *Scratch* as well as *Hold* files, requesting as much space as they legitimately need to do their work. **reapsf** will not delete files that have not exceeded the time thresholds.

AUTHOR

Gareth Loy

SEE ALSO

reapsf(1csound), purgesf(1csound), rmsf(1csound), lsf(1csound), dumpsf(1csound), restorsf(1csound).

NAME

sdc - print map of soundfile disk cylinder usage

SYNOPSIS

sdc [/dev][/user]

DESCRIPTION

sdc prints a map of a *csound* file system's disk cylinder usage. If no argument is given, **sdc** will choose your default file system, with a special indication about which cylinders are owned by you. The argument may be either a soundfile device (/snd or /snd1, currently at CARL), or a user name (preceded with a slash, according to csound program conventions), or both. Examples:

```
% sdc          - lists user's own files on user's default device
% sdc /snd     - lists user's files on device /snd
% sdc /rusty   - lists rusty's files on user's default device
% sdc /snd1/dgl        - lists dgl's files on device /snd1
```

Cylinders occupied by files owned by the person under scrutiny will be printed with '*' and '+' (alternating to show successive files). Unused cylinders are printed with '.', used cylinders with 'u', allocated and freed cylinders with 'a' and 'f'. Allocated cylinders which the owner under scrutiny own are printed as 'A'.

SEE ALSO

lsf(1csound), sfck(1csound), cdsf(1csound), pwsf(1csound).

NAME

sfck - check sound file system for soundness

SYNOPSIS

sfck [flags]

- * f = check connectivity between disk free list and .sdf files
- * z = zap the free list (you must be a member of the disk group)
- n = check and print out cylinder usage statistics
- * F = check connectivity between sound file directory and .sdf files
- * A = do all the above at once, to fix a broken file system.
- p = pretty-print the disk free list file
- y = answer yes to all questions
- * B = boot flag.

Flags preceded by '*' can only be issued by a member of the *disk* group.

DESCRIPTION

sfck is a continuity checker for the sound file system. It can be used to diagnose and fix various illnesses in the sound file system, depending on the flags chosen.

Because of the responsibilities of this program, only members of the *csound* super-user's group can execute certain functions with **sfck.** In all cases, **sfck** will inform you if you are not qualified for a particular operation.

sfck does not masterlock the filesystem it is fixing. This is the responsibility of the user. If running **sfck** on a routine basis uncovers anything more serious than zapping the free list, you should quit **sfck**, run **locksf** on the appropriate filesystem, monitor activity until all files are closed (use ps(1)), then re-run **sfck.**

The Flags and Functions

-n The command: sfck -n is a non-privileged function which prints out the state of the free list, including the following: *maxcyl total used unused alloced freed* where *maxcyl* is the maximum initialized size of the file system, *total* is the sum of all blocks in the actual list, *used* and *unused* show how many are currently occupied. *Alloced* and *freed* refer to cylinders allocated to or freed by (possibly) in-progress processes. *Biggest* refers to the biggest unused contiguous block of cylinders. This is the largest realtime file that can be written at that moment.

-f The *f* flag reads all sdf files in all subdirectories of the filesystem under scrutiny, and checks their cylinder lists for overlaps with other sdf files. It rebuilds the free list from the result.

-z The *z* flag condenses adjacent unused blocks into one block.

-A The *A* flag does a sequence designed to manually fix a broken sound file system. It does the flag sequence of "fzn" automatically, in that order. The strategy is to first scrub the free list with f and z, check to see that these actions haven't done any harm with n. If the free list is scrambled or blocks are missing, duplicated, or otherwise in bad shape, **sfck** goes through and arbitrates conflicts between files for who owns cylinder blocks (or parts of blocks). It votes in favor of whichever file was most recently occupying a block, and requests your permission to delete files that are adversely affected by this decision. It collects a list of all lost files that result from this process in /<filesystem>/loosers. A message should be posted on the system telling these users that they have lost files.

-B The *B* flag is equivalent to the flag sequence fzy in its function. It is designed to be run at boot time.

AUTHOR

Gareth Loy

SEE ALSO

lsf(1csound), rmsf(1csound), mvsf(1csound), visf(1csound).

BUGS

Sometimes blocks can end up being listed as allocated or used on the free list but there is no .sdf file to claim them. This can happen if, for instance, an .sdf file gets zeroed or deleted erroneously through some system error. **sfck** does not reclaim these orphaned blocks when it sees them so as to not clobber the affected cylinder block. This allows a file to be rebuilt by hand in the rare circumstance that that is desirable. If you want to flush such erroneously allocated blocks to retrieve their space, rename /<fs>/dskcyls to some other name and run **sfck**, on that <fs> again. **sfck** will rebuild dskcyls from existing .sdf files, effectively clearing the erroneously "used" blocks.

NAME

sfdt - print status of csound dump regimen

SYNOPSIS

sfdt csound_filesystem_name

DESCRIPTION

sfdt will format and print the status of the *csound* dump regimen for a particular file system. For instance:

```
% sfdt /snd
level  date                        tape id
0      Wed Nov  3 13:49:38 1982      2
5      Thu Nov 11 12:46:31 1982     18
6      Wed Nov 24 10:46:00 1982     19
```

sfdt reads the UNIX file *<csound_filesystem_name>/sfdumpinfo* which is maintained by **dumpsf** for each dump at every level for every file system.

SEE ALSO

dumpsf(1csound), restorsf(1csound), "Managing a Csound Tape Dump Regimen", Gareth Loy, CARL Technical Report, 1982.

NAME

sfnorm - write normalized samples on standard output

SYNOPSIS

sfnorm [flags] [file] > floatsams

flags:

a#	normalize to amplitude #
bN	begin at time N
eN	end at time N
dN	duration is time N
h =	help message

DESCRIPTION

sfnorm produces the amplitude-normalized samples of the named sound file on its standard output. Normalization is only computed for the window specified by the flags.

If the -a flag is given the samples are normalized to that value instead of 1.0.

AUTHOR

Gareth Loy

NAME

show, fshow, yshow – CRT waveform display hack

SYNOPSIS

show [flags] < floatsams > text mode graph

DESCRIPTION

Input must be a file or pipe. flags:

-l set lower bound of display to N (-1)

-u set upper bound of display to N (+ 1)

-m set upper and lower bounds to + and - N ([-1,+ 1])

-R show time instead of sample number using sampling rate N (not available in yshow)

-a display average of N seconds of samples.

-e display mean squared energy of N seconds of samples

-s skip output by N seconds worth of samples of input

-Cc show sample value as character c ('*')

-Hc histogram mode using character c ('-')

-S set window jump size to N (yshow only). All durations are in seconds. Use postop 'S' for sample times. Arguments may be expressions.

The **show** program will crudely plot a waveform on a CRT terminal. If no range arguments are given, samples are expected to lie in the signed unit interval, [-1,+ 1]. The -m flag will set the lower bound to -(upper bound).

fshow is exactly like show, but it uses the **curses(3)** cursor optimization package to make it fast (it's even fast enough on 300 baud terminals!). **yshow** is also like show, but it scrolls the waveform horizontally instead of vertically (for those who don't like to lie down on their sides to view waveforms). It also uses **curses**, but, since it must do software scrolling it is pretty slow.

Ordinarily, **show** displays every sample point given it. This is tedious at best for large quantities of input. You can condense the number of points to be displayed by several methods. The -s flag skips over so many samples of data for each one displayed. It takes an argument specifying time in seconds to skip by. e.g., -s.001 skips by over millisecond's worth of input points for every output point. You can take the average of a window of time with -aN, where again N is time in seconds. Lastly, you can take the mean squared of a window size of time N with the flag -eN. Averaging is good to compress waveforms of very long wavelength. Mean squared is good to view the overall envelope. You have to play with the value of N for each situation.

The number of samples in a window is determined by the sampling rate. The default rate is 49152 samples/second. The default window size is 2.6 milliseconds. Change the sampling rate with -RN, where N is the new sampling rate, e.g., -R16K. Change it first before setting any other time variables.

Window size can also be set as samples duration rather than time by the use of the 'S' postoperator. e.g., -e32S tells **show** to use an averaging window of 32 samples. (Without the 'S' postop, it would be 32*49152 samples in size!) All arguments may be expressions. In particular, operators + - * /, and postoperators S (samples), K (times 1024), k (times 1000), s (seconds, default) and dB are supported.

The -bN, -eN and -dN flags specify the time to start displaying, stop displaying and duration to display, respectively. N is in seconds, at the prevailing sampling rate.

Histogram output can be obtained with -Hc where the character c immediately following the H will be the character used for the bars in the graph. If none is specified, it chooses one. Similarly, you can change the character that indicates sample values by supplying the one you want as an argument to -C.

yshow has an additional flag: -SN, which sets the number of samples to skip over when scrolling.

Don't forget to express lower bound as a minus number if that's what you really intend.

AUTHOR

F. R. Moore (show), Gareth Loy (fshow, yshow).

SEE ALSO

cmpsig(1carl)

NAME

signum - derive sign of floatsam stream

SYNOPSIS

signum [**median** [**gain**]] < floatsams > floatsams

DESCRIPTION

signum reads floatsams (32-bit binary floating point samples) on its standard input and extracts the sign of each sample. By default, the value of **median** is 0, and **gain** is 1. The input signal is first scaled by the value of **gain** then, if the resulting sample is greater than **median**, the output is 1.0; if the input sample is equal to **median**, the output is 0; if the input sample is less than **median**, the output is -1.0.

NAME

sndcmp - compare two sound files.

SYNOPSIS

sndcmp csound_file_1 csound_file_2

DESCRIPTION

sndcmp compares two *csound* files for equality. It checks all channels of the files presented. Where differences are encountered it prints out the absolute sample number, the time in seconds at the sampling rate of the first file, and the two sample values. At the end, it lists the number of differences encountered, or says "no differences" if none are found. If the files are of unequal length, this information is given.

AUTHOR

Gareth Loy

NAME

sndhist - produce histogram of sound file

SYNOPSIS

sndhist [flags] [file]

DESCRIPTION

sndhist is the rather un-UNIX like implementation within one program of the exact same functionality obtained by the combination of sndin(1csound) and hist(1CARL). To wit, saying

```
sndin -b1 -e2 |hist
```

is exactly the equivalent of saying

```
sndhist -b1 -e2
```

APOLOGIA

The purpose of this heresy is also rather un-UNIX-like in nature: speed. For recording sessions, fast estimation of gain levels is essential. By eliminating the pipe, **sndhist** runs better than three times faster than **sndin** and **hist** piped.

The author of the program hopes the gods of UNIX will not frown too darkly upon this sin.

BUGS

The devil made me do it!

AUTHOR

Gareth Loy

NAME

sndin - read csound files onto standard output

SYNOPSIS

sndin [flags] [filename] > output

output format:

if stdout is a tty, writes arabic numbers,
else if stdout is a file or pipe, writes floatsams

flags:

-oX X overrides default output format, X can be
f pipe: floatsams, tty: floating point
s pipe: shortsams, tty: integer

-t force arabic output (even if output is file or pipe)
equivalent to "sndin | btoa -s"

-bN set begin time to N

-eN set end time to N

-dN set duration to N

-cS S is a comma-separated list of selected channel numbers

-H suppress generating a header.

defaults:

begin time = 0,
end time = end of file,
file name = file named test in your current sound file directory.

DESCRIPTION

sndin reads samples from an existing sound file onto its standard output. The format it writes depends on what it is connected to, and any output format flags. If it's standard output is connected to a file or pipe, it writes **floatsams** (32-bit floating point samples), else, when connected to a terminal, it writes the sample number and sample value one line for each sample.

The Flags

All flags for **sndin** take options (except -h and -t), that is, the flag must be followed by a value, with no intervening space.

The -o flag sets the output format, that is, the format for the samples to be written on the standard output. This flag takes an "option", which is how to do the formatting. The two forms of the flag are -of and -os. The 'f' option to this flag is the default. With this default in action, and the output connected to a file or pipe, **floatsams** are written; if connected to a terminal, sample number and floating point format samples are printed. The 's' option causes **shortsams** to be written when the output is a file or pipe; if connected to a terminal, this option causes sample number and integer format sample values to be printed.

The -t flag causes arabic text format to be printed even when the output is connected to a file or pipe. It is equivalent to the following command: "sndin | btoa -s".

By default, **sndin** writes a header on the front of the data. The -H flag strips this off.

Specifying Begin and End Times

Specify the begin time and/or the end time by supplying -b or -e flags, as in:

% sndin -b3

which will read file *test* from 3 seconds to the end of the file, or:

% sndin -e3.1

which will read file *test* from the beginning to 3.1 seconds.

You may express time in samples instead of time in seconds when indicating beginning and ending points by use of postoperators. For instance,

% sndin -b5S -e13ms

reads file test from its fifth sample (truncated to the nearest sample frame boundary), ending at 13 milliseconds. Available postoperators include S = samples, s = seconds (default), ms = milliseconds, m = minutes. The postoperator K = N*1024, and k = N*1000. The -d flag specifies duration instead of end time.

Channel Selection

The -c flag takes a comma-separated list of channels to output. Channels are numbered from 1. The number of channels is set by the number of channels in the file. If the file test were, say 8-channel, we could examine the first ten sample frames of channels 1,3,4 by saying

```
% sndin -e10S -c1,3,4
```

Note that not including a -c channel specification selects all channels.

A variant on the channel specification allows a group of channels to be selected. Saying -cNxM, where N and M are channel numbers, selects all channels starting with N, separated by M channels. For example, – c1x2 selects channels 1, 3, 5, 7, 9... out to the number of channels. This facilitates extraction of channel sets, for instance, of phase vocoder data, where the vocoder writes odd numbered channels as amplitude, even numbered channels as frequency.

There may be no blanks in the list of channels.

AUTHOR

Gareth Loy

SEE ALSO

sndout(1csound).

DIAGNOSTICS

You will be told if the sound file does not exist, or if you may not read it.

BUGS

The window of samples selected is different depending on the presence or absence of the -c flag. With -c absent, all channels are output, and if the the window selected with -b, -e and -d time flags includes the S postoperator, then the S postop. specifies *absolute samples*, (rather than sample frames). With -c present, if time flags include the S postop., then the S postop. specifies *sample frames*. Thus, the command

```
% sndin -e10S
```

prints the first 10 samples of the file. The command

```
% sndin -c1 -e10S prints channel 1 of the first 10 sample frames.
```

NAME

sndout - write sound files.

SYNOPSIS

sndout [config_flags] [file]
or
sndout [time_flags] [file]
time_flags:

-bN	set begin time to N (use $ to mean current EOF)
-eN	set end time to N
-dN	set duration to N

config_flags:

-cN	number of channels: 1, 2, 4, default: 1;
-RN	sampling rate: 0 to 49152, default: 16384;
-i	input packing: s (short), f (float), default: f;
-o	output packing s (short), f (float), default: s;
-TN	file size (expressed as time in seconds)
-CN	file size (expressed in cylinders, default: 1;
-tc	realtime flag: r (realtime), n (not realtime), default: n;
-rtext	remark: quoted text string, default: none;
-Ifile	include file: unix filename, default: none;
-pN	file protection: 0000 - 0666, default: (0644)
-S	set scratch status for file
-H	set hold mode
-K	set keep mode (Note: this is disabled at CARL, see below)
-h	help message, prints a summary of this list.

sndout reads headers for input data format, sampling rate and number of channels. Flags override header properties.

DESCRIPTION

Sndout reads binary samples from the standard input, and writes them in a file on the sound file system.

The Flags

All flags for **sndout** take options (except -h), that is, the flag must be followed by a value, with no intervening space.

The flags for **sndout** are divided on whether they specify how the file is to be configured (config_flags) or where to start/stop writing samples into the file (time_flags). When time_flags are used, it is assumed that you are modifying an existing file, and the config_flags, if any, are ignored.

Config_Flags

The -c flag sets the number of channels: 1 = mono, 2 = stereo, 4 = quad.

The -R flag sets the sampling rate. At CARL, the default sampling rate is 49152 sample-frames per second, to match the native speed of our DSC converters. A sample frame contains one sample for each channel.

The -i and -o flags set the packing mode. Samples appearing on the standard input are expected to be binary floating point numbers (as indicated in the synopsis by the 'f' default for the input packing flag). Similarly, the -o packing flag determines the format of the samples in the sound file being written. By default, samples are stored as shortsams (short integers) for compactness of storage, and so that they may be played through converters in realtime.

The flags -S -H and -K set *scratch*, *hold* and *keep* status, respectively for the new file. Note: at CARL, *keep* mode may not be specified when a new file is created. This is to make it difficult

to automatically bequeath *keep* status on files, since this practice can have antisocial consequences if overused. Use *keepsf(1carl)*.

Comments can be included in sound files by the use of the -r flag. For instance,

```
% sndout -r"look ma, no hands"
```

will add this text as a comment in the file.

The -IX (for "include") flag copies the name of the file given in X into the descriptor file. The *csound* file system itself does nothing with this flag except to carry the filename along in the descriptor file. However, this mechanism allows other programs to associate auxiliary data to a sound file. This could be used, for instance, to keep track of the names of the score files used to create the sound file, or more extensive comments, or the like.

A subset of the unix file protection scheme is implemented. Default protection of 0644 is decoded to mean: 6: owner can read and write the file; 4: owner's group can read the file; 4: others can read the file. 4 gives read permission, and 2 gives write permission to owner, owner's group, and others, respectively. The other protection bits are reserved; there is (currently) no adjustable directory protection. Directory permission is currently hardwired as 0644, meaning only the owner may write in it, but all others may read it.

Contiguous / Non-contiguous Files

By default, a sound file occupies non-contiguous blocks of storage. This allows files to grow on demand. However, there are potential liabilities associated with non-contiguous files, in that the time it takes the disk head to reposition itself from one block to the next may be greater than the time lag the converters can withstand, resulting in clicks or missed data. Generally, for recording, contiguous files are used to help guard against missed data, and for all other purposes, the default non-contiguous files work fine.

Files are created non-contiguous by **sndout** unless the -T or -C or -tr flags are given. If any of these flags appear, the file is made contiguous, and it may not grow in size beyond the space claimed with these flags.

The -T and -C flags set the size of contiguous sound files. The -T flag allows the size of the file to be set in seconds, the -C flag sets it in units of cylinders. In either case, files are measured in units of cylinders, the T flag is just a convenience. A file of one cylinder's length, with short output packing set, mono, at 49152 KHz sampling rate lasts about 3 seconds. (Note that this means that a file that may store just a few samples will still occupy an entire cylinder.)

The -t flag can override the -T and -C flags for the mode of the file. Its use is deprecated.

Time_Flags

Once a file is created, its contents can be overwritten, and possibly extended by use of the time_flags. For instance, if file named "foo" exists and is 1 second long, then

```
% sndout -b.3 -e.7 foo < silencefile
```

will cause the first .4 secs of the UNIX file silencefile to replace the contents of foo. If silencefile is longer than .4 secs, the remainder is not used. If it is less than .4 secs, the remaining samples are unchanged. Saying:

```
% sndout .3 .7 foo < silencefile
```

is equivalent to the example above. Here's another equivalent:

```
% sndout -b.3 -d.4 foo < silencefile
```

A realtime file can be extended out to the limit of its cylinder allocation this way. A non-realtime file can be extended out to the limit of available cylinders. For instance, if sound file test is a realtime file, has 1 cylinder, is mono, using 16 bit samples, at 16KHz sampling rate, it is able to store 9.5 secs of sound. Thus, if it currently stored 1 sec. of sound, the call:

```
% sndout -b1 -e9.5 < silencefile
```

Samples will be added from time 1 sec out to a maximum of 9.5 or the end of silencefile.

Simple arithmetic expressions and the postoperators 's', 'S' and 'K', 'm' and 'ms' are available and work as they do for sndin. For instance, -b1KS specifies beginning at the 1024'th sample frame.

The starting sample you specify may not be greater than the length of the file. For instance, if file test has 16384 samples, the call:

```
% sndout -b16384S < ...
```

will fail, whereas

```
% sndout -b16384S
```

will succeed, and begin concatenating at the end of the file.

A better way to concatenate to the end of a sound file is as follows:

```
% sndout -b$ < samples
```

The '$' operator specifies "the end of the file" and can be used to guarantee concatenating after the last valid sample.

Flags and Headers

sndout reads headers as described in *procom(3carl)*. It is gleans the input data format, the sampling rate and number of channels from the header. If flags are given for these properties, they override the values obtained from the header.

SEE ALSO

sndin(1csound)

DIAGNOSTICS

The descriptor of the file is rewritten for each buffer of sound data written. This allows e.g.,

```
% lsf -f
```

to show you how the file is growing, etc.

NAME

sndpath – create/edit a sound trajectory

SYNOPSIS

sndpath [**-b**] [plotfile] flags:

-b don't print box around edge of screen.

Plotfiles are created with the **P** command, and have .p as a suffix. Leave off the suffix when naming a plotfile.

DESCRIPTION

sndpath is an interactive program to produce and edit two dimensional sound trajectories or paths. It works on ordinary CRT terminals by using the curses(3) screen control package. The user is presented with a rectangle the size of the screen with a '+' sign in the middle indicating the center of the sound space. Four numbers surround the '+' representing the four speakers typically used to reproduce such sound paths. The user marks the locations the sound is to pass through in the order they occur. On command, **sndpath** draws a smooth curve (a cubic spline function) through the points the user has specified, creating the trajectory. The user then writes out the spline function to a file. The file can be in a variety of formats, suitable for digestion by various programs that can control sound distribution in space, such as *cmusic(1carl)*. Commands exist in **sndpath** to modify, append and delete points on the path. The area covered by the screen display can be scaled to represent a range from a few meters to many kilometers.

File Formats

There are three file formats **sndpath** can create files of: 1) binary floating point functions, 2) text functions, and 3) screen plots.

1) Binary floating point function files are written with the *w* command (see below). **sndpath** prompts for a filename, then writes two files: filename.x and filename.y (the .x and .y are added automatically, you just supply the filename part). These two files contain the x and y coordinate values, respectively, of the path, in binary floating point format (called floatsams). In this format, the two files can be read into *cmusic* via the *genraw(1carl)* function handling program for *cmusic*.

2) Text function file format, which is written with the *W* command, contains [x,y] coordinate pairs for each point on the path arranged as one coordinate pair per line. The file is in text format. This format is useful for two particular applications. First, the path can be passed to *graph(1)* and *plot(1)* for rendering on a high resolution printer. Also, this is suitable for another function handling program called *quad(1carl)*, which implements Chowning-style spatial manipulation. Text function files have a .t appended to the name automatically.

3) Lastly, the **sndpath** representation of the screen image can be saved in a plot file. These are written with the *P* command. Plot files contain summaries of the state of the display. A **sndpath** plot can be saved this way such that it can be retrieved by **sndpath** for further editing. There are two ways of retrieving a plot file for editing. First is to name it (without the .p suffix) when starting **sndpath.** Inside **sndpath,** the *r* command reads a plot file previously created with a *P* command. (Again, leave off the .p). The *a* command appends the named plot file to the current contents of the screen.

Commands

q quit sndpath,
? print a help message,
u i o j l m , . move cursor one position,
U I O J K L M < > slam cursor to window edge, K to center,
x [spacebar] put/remove a node where cursor is,

y p yank a node, place a node,
[RETURN] draw spline curve through nodes,
e E erase spline curve, erase whole screen,
f b move forward/ back to next/ previous node,
n interpolate new point between this and next node,
w write xy binary floating point function files
W write text function file
P write sndpath plot file
r a read in plot file, append plot file to screen,
t set time of a node,
: enter extend mode,

Extend mode commands:

z	set set zoom size (2.0),
c	set x,y coordinates of center of terrain displayed (0 0)
d	set distance between speakers in meters (5),
P	print nodes,
o	add offset to path (type two blank-separated numbers)
s	scale path (type two blank-separated numbers)
r	rotate path (type degrees of rotation)
i	toggle spline interpolation mode (reset),
n	set number of total spline points plotted (128 average),
t	set time in seconds to scale duration of entire function,
k	set spline constant (0),
p	toggle spline periodic mode (reset),
T	toggle spline time mode (set)

FILES

There are three kinds of files generated by **sndpath.**

Plot files

These are small text files which contain the state of a plot in ASCII format. They have a .p filename suffix appended automatically. Ingredients of the state which are saved include:

* interspeaker distance,
* zoom scaling,
* number of spline points,
* x, y, and time position for each X point,
* spline mode,
* spline time mode,
* spline constant.

Text x,y files

These are ASCII files of x,y number pairs representing the spline-fitted curve, suitable for input to, e.g., *graph(1)*. No suffix is appended.

Floatsam files

Two files are written, file.x and file.y which contain x against time and y against time respectively of the spline-fitted curve, suitable for input to e.g., *cmusic(1carl)*.

AUTHOR

Gareth Loy

SEE ALSO

cmusic(1carl), quad(1carl), graph(1), plot(1), "Sndpath, a Program to Create/Edit Sound Trajectories", in /mnt/tutorials/sndpath.dgl locally at CARL.

BUGS

While this paradigm works well for most trajectories, there are pitfalls. In particular, the spline function places goodness of fit over everything, including length of the function, so the function length can be off by as many as 10 points.

Spline functions also have other pathological reactions to certain trajectory specifications. In particular, the number of spline points for a trajectory is not automatically adjusted to the path. For instance, if 1000 spline points are specified for a path of 0 - 100 seconds, and if two path points are separated by less than .10 of a second, no spline points will be interpolated between them, and a perceptable discontinuity can occur. The solution is to specify more spline points to be interpolated, or to readjust the trajectory.

NAME

spect - apply FFT to floatsam stream

SYNOPSIS

spect [**flags**] < floatsams > output

flags: (defaults in parenthesis)

-p	output power spectrum as floatsams
-d	output power spectrum in dB as floatsams
-c	output complex spectrum as floatsams
-a	output average power spectrum (over -n iterations) in dB
	(if none of above, a crude CRT plot of the spectrum is written)
-RN	set sample rate to N (default is read from stdin)
-bN	begin at time N (0)
-wN	window duration for FFT set to N seconds (1024S)
-nN	number of iterations: FFT N successive windows of input data (1)
-sN	skip window by N seconds between successive FFT's (-w / 2)
-lM	use M pole linear prediction estimate of input (autocorrelate)
-kM	use M pole linear prediction estimate of input (covariance)
-r	rectangular window: suppresses default hamming window
-u	un-normalize: suppresses default normalization of FFT to 0 dB
-f	filter evaluation: input is filter impulse response

or

spect [-p, -c] **sample_offset window_size N_windows** < floatsams > output

where **sample_offset** corresponds to -s, **window_size** corresponds to -w, and **N_windows** corresponds to -r.

DESCRIPTION

spect reads floatsams (32-bit binary floating point samples) on its standard input and takes (possibly) successive FFTs of the signal.

There are two modes of usage, with flags, or with fixed arguments. If you use fixed arguments, all three arguments must be supplied in the order specified in the second usage description given above. On the other hand, flags may be given in any order, and any omitted will be given default values as shown. One possible source of confusion is that the fixed arguments refer to durations in terms of number of samples while the flags refer to duration in terms of number of seconds (unless the postop "S" is used). This discrepancy exists for historical reasons; the flags are very strongly preferred.

There are two distinct formats that *spect* produces. By default, the output is a crude crt graphics plot of the power spectrum plotted from 0 to the Nyquist rate (i.e., half the sample rate). If the -p flag is given, the power spectrum only is output as a floatsam stream. If the -c flag is given, the complex spectrum is written as floatsam pairs, real, then imaginary in sample interleaved form. If the -d flag is given, the power spectrum only is output as a floatsam stream, but in decibels. This form is highly suited for piping to the standard UNIX graphics filters. For example:

```
sndin file |spect -d [flags] |btoa -s |graph -y -100 0 |v550
```

If the -a flag is given, the power spectrum only is output as a floatsam stream in decibels, but only after all n iterations have been completed and the n indiviual spectra are all averaged together. This is extremely helpful when looking at noisy inputs. If the skip is set to half the window size (the default), then the classical periodogram method for power spectrum estimation results. (See the helpfile for *pmpse*.)

If the -l or -k flag is specified, then *spect* displays the spectrum of the linear prediction estimate of the input instead of the true spectrum. The linear prediction spectrum tends to be more like a plot of the spectral envelope than the actual spectrum. This means that individual harmonics do not show up in the display; only broad resonances are visible. There are two different methods of linear prediction which may be specified: the -l flag calls for the autocorrelation method, and the -k flag calls for the covariance method. For very short windows, the covariance method is preferred, but for windows greater than 256 samples the results are nearly identical for each. Both the -l and -k flags must be followed by a number which specifies the number of filter coefficients to be used in the linear prediction model. A value of 16 is suggested for speech. (See the helpfile for *lpc*.)

The remaining flags are intended to be fairly obvious. The most critical is -w because the FFT size (and spectral resolution) will be directly proportional to the duration of the window. On the one hand, the window should not be so long that the signal changes its character within the duration of the window. On the other hand, the window must be long enough to provide adequate frequency resolution. For example, if the sample rate is 16KHz and the input signal has a pitch of 64Hz, then -w will probably have to be at least 2048S (this gives a frequency resolution of 16KHz / 2K = 8Hz) in order to clearly see the peaks of each separate harmonic. The default window is a hamming window (w(i) = .54 + .46 * cos(2 * pi * i / (n/2-1)) where i ranges between plus and minus n/2 - 1). The -r flag can be specified to force the use of a rectangular window.

The output spectrum is normalized so that the peak value is at 0dB. This automatic normalization can be suppressed with -u (un-normalize) flag. In this case a sine wave of amplitude 1. will appear as a spectral peak of -6dB (as it should!). The -f flag is a combination of -r , -u , and an additional gain factor so that the spectrum of a filter impulse response can be accurately calculated.

AUTHOR

F. R. Moore & Mark Dolson

DIAGNOSTICS

If the window size would exceed 32K samples, the diagnostic "You gotta be kidding!" is printed on your screen and the program exits.

NAME

srconv - convert signal sampling rates by any positive rational number

SYNOPSIS

srconv l m filterfile < floatsams > floatsams

DESCRIPTION

The sampling rate converter is for changing the sampling rate of a digital signal. The input and output format is **floatsams**, (32-bit binary floating point samples) regardless of whether the input or output is a file or pipe.

The output sampling rate is **l/m** times the input sampling rate, where **l** and **m** are arbitrary positive integers. It is desirable that **l** and **m** have no common factors, as this needlessly adds to the computational complexity. Also, it is better to pipe several stages of *srconv* together using small **l** and **m** rather than use a single stage having large **l** and **m** when **l** and **m** are composite numbers; for example, for **l/m** = 4/9, one can say

```
srconv 2 3 % < test.dat |srconv 2 3 % > out.dat
```

The **%** sign is used to instruct *srconv* to use pre-designed default lowpass filters. The default filters generally have 0.1dB passband ripple, and -60dB stopband rejection. In addition, saying **%%** for the default filter specification uses slow-and-clean default filters. Use the single **%** for quick-and-dirty sample rate conversion, and **%%** for more critical examples. Filter files exist for all values of **l** and **m** for ratios from 1 to 8 for both quick-and-dirty and slow-and-clean default filters.

DIAGNOSTICS

The program interactively requests parameters and filenames when arguments are missing from the command line.

AUTHORS

J. O. Smith and F. R. Moore

FILES

Default filter files at CARL: /usr/local/lib/srconv/*.flt.

BUGS

The implication that single **%** is somehow less than wonderful in the quality of the resulting signal has been challenged by some. It seems that single **%** has been found to be perfectly acceptable for such applications as converting from 48K to 16K files, for instance.

NAME

step – step function generator for cmusic

SYNOPSIS

step – LN x0 y0 x1 y1 ... xN yN

where *N* is e.g. 1024 for a 1024-point output function.

DESCRIPTION

step creates a simple step function from the arguments given.

The *-L* value specifies the number of points on the function. Successive [*x*,*y*] pairs specify that the function is to adopt value *y* at point *x* and repeat it for subsequent *x'es* until a new *x* point is reached. The function is terminated by the value of xN equaling the value given with *-L*.

Example line in a *cmusic* score:

```
gen 0 step f1 0 0 1 10 5 256 .3 384 0 1023;
```

Example usage from a shell:

```
% gen -L32 0 1/2 3 5 13 .33 25 1
```

in the case where the last point specified is less than the last point in the function, the last value specified is copied across the remaining points. In the case where the points specified are greater than the last point of the function, they are ignored.

If standard output is a terminal, ascii values of results are printed, if a file or pipe, floatsams (floating point binary samples) are written.

SEE ALSO

gen0(1carl), gen1(1carl), ... gen6(1carl), chubby(1carl), cspline(1carl), and cmusic(1carl).

NAME

stochist – plot histogram of stochastic function on CRT terminal

SYNOPSIS

stochist [-lN] [-uN] [-nN] [-f] < floatsams > text or floatsams

DESCRIPTION

Input must be file or pipe of floatsams. If output is a terminal, a text histogram is produced. If output is file or pipe, histogram sums are written as floatsams. flags: (default)

-lN set lower bound of display to N (-3.0)

-uN set upper bound of display to N (3.0)

-nN set number of histogram windows to N (22)

-f force writing text bar graph

stochist plots a histogram of an arbitrary function on a CRT terminal. It is a convenient way of observing, e.g., the output of *cannon(1carl)*. It's function is specifically for viewing stochastic functions, not for observing the behavior of signals. To generate histograms of signals, see *hist(1carl)*

Example usage:

% cannon gauss !stochist

The operation of **stochist** is to construct a grid of numerical ranges, called windows, and to compare the numbers read in from the standard input against those windows, maintaining a count of how many numbers fall into each window. When the standard input is exhausted, a display of the number of hits in each window is produced on the standard output in the form of a horizontal bar graph.

The number and extent of the grid ranges is determined as $\frac{u-l}{w}$ where u is the upper bound, l the lower bound, and w is the number of windows. There will be one bar in the resulting graph for each w. (w defaults to 22, so the display will just fill the screen on a 24-line display).

When the output of **stochist** is a file or pipe, the numerical value of the window counts are output instead of the bar graph representation.

Some statistics are kept on the histogram which are printed as a line after the bar graph. These values are: the actual minimum and maximum values read in, the value and location of the mode (window with the greatest number of hits), the mean, and the number of samples, respectively.

AUTHOR

Gareth Loy

SEE ALSO

cannon(1carl) hist(1carl)

DIAGNOSTICS

Input must be a file or pipe. This and any error in input flags generate a usage statement.

NAME

stripheader – remove a header from a floatsam stream

SYNOPSIS

stripheader < floatsams > floatsams (no header)

DESCRIPTION

stripheader reads samples from *stdin*, removes any header found there, and writes the headerless samples on its *stdout*.

AUTHOR

Gareth Loy

SEE ALSO

readheader(1carl)

NAME

sun2vax - transmorgrify SUN floating point to VAX format

SYNOPSIS

sun2vax < sun_floatsams > vax_floatsams

DESCRIPTION

sun2vax reads floatsams (32-bit binary floating point samples) on its standard input in SUN format, and converts them to VAX format.

This allows binary floating point data generated on a SUN to be transferred to a VAX.

DIAGNOSTICS

This routine takes no arguments, and reads and writes only floatsams. Either giving it arguments, or leaving its standard output connected to a terminal will generate a usage statement. It passes headers correctly.

BUGS

This program only works on a VAX. There should be a corresponding program to VAXify floatsams on a SUN. Generally, all programs should write a header property saying what specific floatsam format the data is in, and this program should test that property before conversion.

SEE ALSO

vax2sun(1carl).

AUTHOR

Gareth Loy

Running **locksf** by itself generates a terse usage message.

NAME

vax2sun - transmorgrify VAX floating point to SUN format

SYNOPSIS

vax2sun < vax_floatsams > sun_floatsams

DESCRIPTION

vax2sun reads floatsams (32-bit binary floating point samples) on its standard input in VAX format, and converts them to SUN format.

This allows binary floating point data generated on a VAX to be transferred to a SUN. It passes headers correctly.

DIAGNOSTICS

This routine takes no arguments, and reads and writes only floatsams. Either giving it arguments, or leaving its standard output connected to a terminal will generate a usage statement.

BUGS

This program only works on a SUN. There should be a corresponding program to SUNify floatsams on a VAX. Generally, all programs should write a header property saying what specific floatsam format the data is in, and this program should test that property before conversion.

SEE ALSO

sun2vax(1carl).

AUTHOR

Gareth Loy

NAME

visf - edit sound file parameters

SYNOPSIS

visf [-h] [-w] soundfile

flags:
-h print help message first
-r readonly mode
-w wizard mode access

DESCRIPTION

visf allows you to alter selected parameters of a sound file. The action of the program is to place you in the vi editor with parameters from the sound file you specify.

The parameters are presented one to a line. The first thing on the line is a code for what the parameter means, which is followed by the actual parameter. You can alter any of these parameters to any value you know is ok, or add or delete comments or include files. Numeric values must be constants — they may not be expressions.

When you are done, quit vi with :x[RETURN] as usual. If you have specified the -r flag, **visf** will immediately exit without recording any modifications you have made. Otherwise you will be asked whether you wish to write out the changes, abort, or get help. 'x' aborts, '?' prints a help message, and then puts you back in vi for another chance to edit the file. Just [RETURN] finishes by writing out your changes and exiting. If you run **visf** with the -h flag, you get the help message first thing, then everything is as above.

The meaning of the parameter codes is as follows:

p file protection code (default = 0644)
R sample rate (range = 64 to 49152) (default = 16384)
P packing mode (s or f) (default = s)
c number of channels (1,2 or 4) (default = 1)
number of samples in file (you can only make this number smaller!)
r comment string, there may be any number of comments, but
r each must begin with "r " in the first column position.
I include file
I and there may be any number of include files, here is a real example:
I ~dgl/ready.seg

You can modify, add or delete comments, change include files, alter the sampling rate, etc. Care must be exercised to be sure that you do not delete lines critical to the operation of the sound file system, such as sampling rate, etc. The correct code letter must appear in the first column and be immediately followed by one space. Note: this is a slightly dangerous program! It is up to you to do it right! Havoc can result from misuse.

In addition, if you are root or a member of the disk group and specify the -w flag, **visf** will display and allow you to modify these parameters as well:

t realtime flag (r or n)
f file name (don't change, use mvsf!)
v cylinder allocation blocks (careful!)
w creation date
x last referenced date
y last altered
z date dumped
k tape key

Use of the -w flag allows you to grow the file size to any value.
This value should not exceed the maximum number of samples available
for the number of cylinders claimed by the file.

DIAGNOSTICS

The program exits if the file is not there or is unreadable, or if you specify the -w flag and are not a superuser.

If you are not the owner of the file, it tells you it is using read only mode, but allows you to look at the file. When you then exit, it tells you the file is unchanged, and exits immediately.

Unknown parameter codes or garbage parameters cause all modifications to be aborted entirely.

BUGS

This is a very gullible program, and far too trusting of your good intentions. Bulletproofing it would take a lifetime, however.

In general, it is up to you to know what to do with the parameters. Check with a wizard before getting too experimental, but there is little you can do that should cause you to come to grief (unless you use the -w flag and are careless). Caveat emptor.

AUTHOR

Gareth Loy

SEE ALSO

sndin(1csound), sndout(1csound), lsf(1csound).

NAME

wave - generates simple test tones on the standard output

SYNOPSIS

wave [flags] > floatsams

flags:
(N is a number or arithmetic expression, C is a character)
-RN = set sampling rate to N (16384 Hz)
-fN = set frequency of waveform to N (440.0 Hz)
-wC = select waveform C from following list:
s = sine (default)
r = ramp
q = square
t = triangle
p = pulse train
i = impulse response
d = dc, where -aN determines the offset
S = silence
-aN = amplitude N, where N may be
absolute amplitude in the range 0 to 1.0,
or 0dB to -90dB (you must supply "dB" postoperator)
-TN = set waveform duration to N (1 second at prevailing -R)
-H suppress generating a header.
-SN append N seconds of silence at end of waveform.
Flag options may be expressions which can include postoperators.
Postoperators: S = time in samples, K = "times 1024", k = "times 1000", s = time in seconds (default postoperator for time values), ms = milliseconds, m = minutes, dB = amplitude in dB.

DESCRIPTION

wave writes simple waveforms on stdout according to specifications supplied by the flags. The output data stream normally consists of binary, floating-point sample numbers in the range of -1.0 to +1.0. If the output is connected to a terminal, the samples are presented in human-readable form instead of binary.

Waveforms available include: sine, square, triangle, sawtooth, pulse train, dc offset, silence and impulse. They are selected with the -w flag. For instance,

% wave -wr

selects a ramp waveform.

Amplitude is set with the -a flag. Amplitudes can be expressed either as a magnitude between 0 and 1.0, or in dB, where 0dB is equivalent to a magnitude of 1.0. These two examples are equivalent:

% wave -a.1

% wave -a-10dB

Note that in the illustration with dB, the second '-' is taken to be a minus sign.

Similarly, *frequency* is set with -f followed by a number, sampling rate is set with -R and a number, likewise for duration.

By default, wave produces a sine wave at A440, sampling rate 16384 FOR one second, at full amplitude. That is, if the defaults were made explicit:

% wave -ws -R16384 -f440 -d1 -a1 > file

and

% wave > file

are equivalent. However, if you just say wave all by itself, and do not direct its output to a pipe or file, you will be given a terse help message, explaining all the flags and options.

The options to the flags may be arithmetic expressions. (It is advisable to put expressions inside double quotes, otherwise the shell may think it is supposed to evaluate them). Expression postoperators 's' and 'K' are available. For instance, -R48K sets the sampling rate to "48*1024" = 49152. The time flags interpret duration in seconds by default. If instead you want duration measured in samples, use the 's' postoperator. For instance, -d500S will set the duration to be 500 samples long, regardless of the sampling rate.

AUTHOR

Gareth Loy

NAME

window - applies an envelope to a floatsam stream

SYNOPSIS

window duration rise fall < floatsams > floatsams

All times are in samples

DESCRIPTION

window reads floatsams (32-bit binary floating point samples) on its standard input and applies a simple trapezoidal envelope to the stream.

All times are specified in samples.

AUTHOR

F. R. Moore

BUGS

This should also do Hamming windows too.

SEE ALSO

janus(1carl).

NAME

xform - transform sample data streams

SYNOPSIS

xform [flags] [expressions] < sample_data > sample_data

flags:

-bN set begin time to N
-eN set end time to N
-iX set input format to X
-oY set output format to Y
formats for X and Y:
b arabic numbers are treated as amplitude in dB
d arabic decimal numbers
e expressions in expr() format (input only)
f arabic floating point numbers
o arabic octal numbers
pN set integer conversion scaling to N
s binary short integer (16-bit) numbers
x arabic hexadecimal numbers
(floatsams are read and written by default)
-t index samples with time in seconds
-RN set sampling rate to N
-s index samples with sample numbers
-h usage statement
-# output debugging information

If input mode specifying *floatsams* or *shortsams* is set, input must be a file or pipe. Otherwise, input may be a terminal. *Floatsams* are written if output is connected to file or pipe, unless if an *arabic* or *shortsam* output format is specified.

DESCRIPTION

xform is a relatively general anything-to-anything converter for signals. It can:

* read either floatsams, shortsams, or arabic numerals,
* write either floatsams, shortsams, or arabic numerals,
* handle variable precision integer formats, and
* apply expressions to the sample stream;
* arabic numerals may be decimal, octal, hexadecimal or dB.

Its operation without flags simply copies sample data from its input to its output, subject to format conventions described next.

Output Format Conversion

If the standard output is a terminal, arabic (i.e., humanly readable) floating point is written, otherwise when the output is a file or pipe, *floatsams* (32-bit binary floating point sample data) are written. This can be modified, to force output of *shortsams* (16-bit binary short integers) or various arabic numeric formats instead of *floatsams*, even if the output is a file or pipe.

The various arabic formats are specified by providing output flags of the form *-oY* where *Y* is one of the characters *b* (dB) *d* (decimal) *f* (floating point) *o* (octal) or *x* (hexadecimal). Note that arabic output conversion formats are cumulative: requesting more than one is possible, and will result in their being printed side by side on the same line. (Supply multiple output format flags to do this).

To write *shortsams*, provide the flag *-os*. Specifying *shortsams* for output turns off printing any arabic format. This flag will be ignored if the standard output is connected to a terminal. Because it is not desirable to try to print either *floatsams* or *shortsams* on a terminal, these modes are automatically disabled when the standard output is a terminal. In this case, *floatsam*

input will produce arabic floating point format on output, and *shortsam* input will produce arabic decimal format on output.

Arabic format output can be accompanied by the sample number (flag: *-s*), and/or the time in seconds (flag: *-t*), at some sampling rate (flag: *-RN*, for sampling rate N).

Input Format Conversion

The standard input reads *floatsams* by default when connected to a file or pipe. When connected to a terminal, it defaults to expecting arithmetic expressions in **expr(3carl)** format (see below). Input can be coerced to be any of the same formats as are available for output, as listed above, by providing a flag of the form *-iX* where *X* is, again, one of *b, d, f, o, s,* or *x*. (Again, input of *floatsams* or *shortsams* from a terminal is neither possible nor particularly meaningful, unless you are a computer).

Data Ranges

Floatsam data are nominally expected to range over the signed unit interval (that is, the range [-1, +1]). *Shortsam* data are nominally expected to range over their maximum precision. For instance, 16-bit binary data can represent numbers in the interval [-32768, +32767]. When converting from *floatsam* to *shortsam* data, scaling must be performed. In the typical case, the signed unit interval of floating point data is mapped by default into the 16-bit integer range [-32768, +32767]. Thus when *shortsams,* are read, their values are divided by 32767 so as to map the integer values into the signed unit interval. If the output is *floatsams,* these converted values are output directly. If the output is *shortsams,* they are rescaled by being multiplied by 32767.

The various arabic formats are similarly scaled, to maintain symmetry. Arabic floating point data is expected to be in the signed unit interval, and decimal, octal and hexadecimal are are expected by default to range over the 16-bit binary range of [32768, +32767].

The range of integer data can be modified. The format specification *pN* sets the range (precision) of integer data formats. For example, the flag *-op2047* sets the integer output range to [-2048, +2047], appropriate for 12-bit *shortsam* format. The precision flag can be set to anything, in fact, limited by the precision of the host computer. However, please note that precisions larger than 16 bits will only work with arabic input and output formats. Setting it to a value greater than the precision of 16-bit binary data will not work if the input or output is *shortsams,* because *shortsams* can't represent anything bigger than the interval [-32768, 32767].

Examples

```
(xform reads floating point arabic, writes the same)
        % xform
(xform reads/writes floatsams)
        % wave |xform |sndout
(xform reads floatsams, writes arabic)
        % wave |xform
(xform reads arabic, writes floatsams)
        % xform |sndout
(xform reads/writes floating point arabic)
        % xform -if -of < file > other_file
(xform drives unix plotting programs)
        % wave -T.01 |xform -of |graph -a |plot -T4014
```

Expressions

xform also has the ability to apply arbitrary arithmetic expressions to the input sample stream. Multiple expressions may be given, with each being applied to the result of the last expression in a cascade fashion. The output sample stream is the cumulative effect of all expressions. The expression format is nearly identical to that of **cmusic.**

The current input sample is available in expressions as the symbol *x* (or *X*, upper case equivalent). Thus, the minimal equation to *xform* is simply *x*, which merely copies the inp sample to the output, just as does specifying no expression at all.

Note, the value of the sample can be expected to be in the signed unit interval (unle modified by input format conventions).

A history of the last 1024 input and output samples is kept in a circular buffer. These are ava able in expressions via the special symbols: *xM* and *yN*, for past input *M* and past output respectively, where *M* and *N* are integers giving the sample order. Thus, *x0* is the same as which refer to the current input sample, *x1* is the immediately preceding input sample, and on. Likewise, *y1* refers to the last output sample, etc. Thus, rather arbitrary filtering of signa is possible.

Here are some simple examples (comments in parenthesis):

(Copy input to output).
```
% wave |xform
```
(Same as above).
```
% wave |xform "x"
```
(Offset).
```
% wave |xform "x+ .5"
```
(Differentiate).
```
% wave |xform "x-x1"
```
(Produce sine wave the hard way).
```
% wave -wr |xform "sin(x*2*3.14)"
```
(Averaging filter).
```
% wave |xform ".5*x-.5*x1"
```
(Multiple expressions, output value is 3)
```
% wave -wd |xform "x+ 1" "x+ 1"
```
Note, the "" marks around arithmetic expressions keep the shell from trying to interpr characters such as '*'.

Format of Expressions

Expressions may contain several types of operands and operators. Expressions must not includ blank spaces. The syntax of expressions is a subset of that available in **cmusic**. The current li of possibilities is as follows:

OPERANDS:

x or xN or XN — refers to input sample x, or N'th previous input.

yN or YN — refers to N'th previous input.

numbers — Numbers may have three bases; all are of type float whether they include decimal point or not.

decimal — Any string of digits, which either includes a decimal point, or which does no include a decimal point.

hexadecimal — If a number does not include a decimal point and starts with the characters 0x then base 16 is interpreted.

octal — If a number does not include a decimal point and starts with the digit 0, base 8 will be interpreted.

OPERATORS:

PARENTHESES

Parentheses must balance, and may be used freely to establish operator precedence. Function arguments should be enclosed in parentheses.

UNOPS

The following are unary operators available in expressions in order precedence, with the first set of operators done before anything in the second set. Unary operations are done before binary operations, and binary operations are done before post operations.

{sin,cos,atan,ln,exp,floor,abs,sqrt,rand}

(These are the standard trigonometric functions sine, cosine, and arctangent, from the UNIX math library, as well as the natural logarithm, exponential, floor, absolute value, and square root functions. Rand is a function which returns a random value between 0 and its (positive) argument.)

{-} (Unary minus, as in -3*p5)

BINOPS

The precedence is as shown below: ^ and % are done before * and /, and * and / are done before + and -.

{^,%} (3^.5 means 3 to the .5 power; 397%17 means 397 modulo 17)

{*,/} (5*79.2 means 5 times 79.2; 9/5 means 9 divided by 5 (float result))

{+ ,-} (3+3 means 3 plus 3, 3-8 means 3 minus 8)

POSTOPS

Post operators are done last. They generally modify the resulting value of the expression which precedes them.

dB converts dB (logarithmic) to linear scale example: -6dB = 10^(-6/20) = 0.5 (approx.).

K converts K to units example: 8K = 8 * 1024 = 8192.

Deg converts degrees to radians example: 180Deg = (180 / 360) * TWOPI = 3.14159.

IS computes the sum of the first N inverse terms, i.e., 3invs = 1 + 1/2 + 1/3. 0IS = 0 by definition.

AUTHOR

Gareth Loy

SEE ALSO

The expression format is a subset of that in **cmusic.** Not all things available there are necessarily available here. Expression evaluation is provided via the subroutine **expr(3carl).**

BUGS

For convenience, the notation *y*, which refers to the current output sample (i.e., the one not yet computed) is supported; its value is defined to equal the current input sample. The notation *y0* is undefined, and may produce garbage.

For simple format conversion, it replaces **btoa, atob,** and **seefloat.** However, the implementation of the expression interpreter is grotesquely inefficient for sample data computation. Thus, it does not replace any of the other CARL transform programs, because of its efficiency limitation. In its defense, one can suppose that the time it takes to implement a dedicated program to accomplish some transform will be greater than the running time of **xform** itself.

NAME

zdelay – variable index interpolating delay

SYNOPSIS

zdelay [-DN] [-zN] [-fX] [-dN] [-RN] < floatsams > output

DESCRIPTION

flags: (default)

-DN set maximum delay to be used to N

-zN set initial delay to N

-fX read floatsam function file X

-dN set duration of run to N

-RN set sample rate to N

zdelay can delay a signal on its standard input by a variable function of time specified with the *-fX* flag, where file *X* contains a delay function. The values of delay can specify non-integral indexes. The output sample is then derived from the two adjacent actual samples by linear interpolation.

The delay function must consist of floating point binary data (floatsams) of Y values only (x values are automatically assumed to go from 0 incrementally). Suitable functions can be created with any *cmusic gen* program. The function is interpolated over the number of samples specified with the *-dN* flag, where *N* is the duration in seconds. The function specifies the derivative of the ratio of the output sample increment rate to the input sample increment rate. So for example, values of the function are interpreted as follows: $f(x)=1.0$, output rate = input rate, frequency of signal is unshifted; $f(x)<1.0$, frequency drops; $f(x)>1.0$, frequency rises. For $f(x)=.5$, the frequency drops 1 octave, for $f(x)=2.0$, it rises an octave. This makes the zdelay very suited for implementing doppler shift.

Care must be exercised that the delay does not become positive (i.e., attempts to read future inputs) or too negative (i.e., exceeds the length of the delay line specified with the *-D* flag). In both cases, the sample algorithm "wraps around" in time to the oldest or newest samples, respectively. The *-z* flag can be used to position the initial delay time (effectively a constant time offset) in a place within the delay line by hand. Otherwise, the offset is automatically set to 1/2 the delay line length.

Other (among numerous) uses of a zdelay are *flanging* and *phasing*.

AUTHOR

Gareth Loy

BUGS

This program is virtually obsolete. See the *zdelay* unit generator in *cmusic*.