DIGITAL SIGNAL PROCESSING LIBRARY

USER MANUAL

Version 1.0

Sonitech International Inc
83 Fullerbrook Road
Wellesley, MA 02181 , USA

TABLE OF CONTENTS

DIGITAL SIGNAL PROCESSING LIBRARY

1.0 OVERVIEW

1.1 Introduction

The Digital Signal Processing Library (DSPL) provides a comprehensive set of digital signal processing (DSP) and utility functions in order to provide high-level language support tools for real-time applications development. It is written in the 'C' programming language, which is widely accepted as the high-level language of choice for real-time digital signal processing applications development. 'C' compilers are available for many leading floating-point digital signal processors. DSPL is portable to most of the leading commercially available compilers for signal processors and computer systems.

DSPL is compiled under Microsoft C 5.0.

1.2 System Requirements

DSPL conforms to the requirements of Microsoft 'C'. It is assumed that you are familiar with Microsoft 'C' compiler and the 'C' language. Test program source code is provided; these programs can guide you to use these functions in your applications.

DSPL (Library)

The minimum requirements for using the DSPL (library) on the IBM PC or PS/2 are:

- a Microsoft or compatible C compiler
- 256 K memory
- 2 double sided disk drives
- DOS 3.0 or later

DSPL (Source Code)

DSPL source code may be compiled on any standard (Kernighan & Ritchie) 'C' compiler.

1.3 Registration and Warranty

The registration form and warranty information is given in the back of this manual. We encourage you to return the completed registration card.

1.4 System Installation

Make a backup of all the disks in this package and store the original disk in a safe place. Disk copy can be done using the DOS command DISKCOPY (refer to your DOS manual for more information on this command).

The 'README.DOC' file on Disk 1 contains installation information.

DSPL is provided for all the four Microsoft C 5.0 models: (LIBRARY diskette)

spl	- small model
splc	- compact model
splm	- medium model
spll	- large model

1.5 Test Programs

Test programs are provided for each library routine and in some cases several library programs are called from a single test program. Listings of these test programs are provided in the Appendix.. Source for the test programs is provided on diskettes included with this package (TEST PROGRAM_A and TEST PROGRAM_B diskettes).

1.6 Finding the Source

If you ordered DSPL with the source code option, the source code is provided on a separate diskette (SOURCE diskette). All C source files end in the file extension '.c'.

2.0 DSPL DESCRIPTION

2.1 Module Classification

DSPL modules description is provided in the LIBRARY DESCRIPTION section of this manual.

DSPL has four major classes of modules:

Initilization Modules (I):

1) Modules that are generally called during the initialization portion of a real-time application. An example is the generation of a table of "twiddle factors" used later on by a Fast Fourier Transform algorithm.

Real Time Modules (R):

2) Modules that are called at regular intervals during the execution of a real-time application, and therefore are time-critical. An example is a Fast Fourier Transform algorithm.

Simulation Modules (S):

3) Modules that are called at regular intervals, but are not time-critical. Such functions are generally used to test out a real-time application, but do not form part of it. An example is the generation of additive noise to verify the performance characteristics of a modem.

Utility Modules (U):

4) Modules that are used to build and analyze signal processing algorithms. An example is a routine that plots the poles and zeros of a rational transfer function.

Each module in this library is classified using the above categories. The category is shown parenthetically immediately after the module name;

(R) stands for real-time module,
(I) for initialization module,
(S) for simulation module
(U) for utility module.

Applications written with the above classification in mind generally run more efficiently even though such a classification may not be essential for non-real time applications. Thus, DSPL reduces the gulf that separates real-time applications development and non-real time design and simulation to develop and evaluate signal processing algorithms. It should be noted that the above classification is not rigid; thus a routine such as a random number generator, denoted as (S), may in fact be used for dithering in a PCM application. In this case, the appropriate classification is (R). However, modules denoted as (S) in DSPL are generally not optimized to minimize execution time.

The modules denoted by (R) represent a wide cross-section of instruction types used by real-time signal processing algorithms and applications. They include FFT algorithms (where indexing and floating-point arithmetic are the main operations), system identification algorithms (where numerical ill-conditioning sometimes occurs; thus testing the dynamic range of the processor), Viterbi decoding (a dynamic programming algorithm, where comparisons, conditional branches, and indexing operations predominate), and the decoding of error correcting codes (where integer arithmetic, bit-wise operations, indexing operations, and conditional branches are extensively used). Thus, the processing time for these functions will serve as benchmarks against which the performance of different signal processors and their associated 'C' compilers may be compared.

For each function, the DSPL library reference has the following format:

Module name:
Inputs:
Outputs:
Processing:
Examples:
Related Modules:

The following is a list of functions with a brief description of each:

acorr.c (R) - Routine to obtain the autocorrelation sequence of a data record.

acs.c (R) - Routine to implement the add-compare-select logic of the Viterbi Decoder.

burg.c (R) - Routine to obtain the predictor coefficients of a data record directly using the maximum Entropy Method (MEM).

ccep.c (R) - Routine to calculate the complex cepstrum of a data record.

cencd.c (R) - Routine to generate the encoded symbol stream using a convolutional encoder.

chlsky.c (R) - Routine to solve the autocorrelation normal equations using Cholesky decomposition.

cosft.c (R) - Routine to calculate the Discrete Cosine Transform (DCT) of a real-valued data sequence.

cov.c (R) - Routine to obtain the covariance matrix of a data record.

durbin.c (R) - Routine to solve the autocorrelation normal equations using Durbin recursion.

encd.c (R) - Routine to encode an information vector to obtain the code vector.

elcf.c (R) - Routine to find the coefficients of the error locator polynomial.

erlc.c (R) - Routine to find the roots of the error locator polynomial.

expdev.c (S) - Routine to generate an exponentially distributed random variable.

ffadd.c (R) - Routine to add two elements of a finite field.

ffmul.c (R) - Routine to multiply two elements of a finite field.

fft.c (R) - Routine to calculate the Discrete Fourier Transform (DFT) of a complex-valued data record.

gasdev.c (S) - Routine to generate a Gaussian distributed random variable.

gentwd.c (I) - Routine to generate a cosine table that is used by most "transform" routines.

genwin.c (I) - Routine to generate commonly used "windows" for spectral analysis.

genzec.c (I) - Routine to generate the Zech's logarithms of a finite field.

gshf.c (I) - Routine to generate a table of shifted versions of a generator polynomial.

icosft.c (R) - Routine computes the Inverse Discrete Cosine Transform (IDCT) of a real-valued data record.

irfft.c (R) - Routine computes the real valued Inverse Discrete Fourier Transform (IDFT) of length N given its first N/2 + 1 complex-valued Fourier components.

itwofft.c (R) - Routine computes the real-valued Inverse Discrete Fourier Transform (IDFT) of length N given its first N/2 + 1 complex-valued Fourier components.

laguer.c (U) - Given the degree m and the degree m+1 complex coefficients of a polynomial, this routine improves the root of the mth degree polynomial using Laguerre's method.

mmult.c (R) - Routine multiplies N matrices after finding the best order in which to do the multiplications.
orddit.c (R) - Routine to obtain half-tone images from gray-level images using ordered dither.
ran1.c (S) - Routine to generate a uniformly distributed random variable.
rbit.c (S) - Routine to generate a random bit sequence using a maximum length shift register.
rfft.c (R) - Routine to compute the first N/2 + 1 complex components of the Discrete Fourier Transform (DFT) of a real-valued data sequence of length N.
sinft.c - Routine to compute the Discrete Sine Transform (DST) of a real-valued data record.
synd.c (R) - Routine to compute the syndrome from the received vector.
twoddct.c (R) - Routine to compute the two-dimension Discrete Cosine Transform (2-D DCT) of a real-valued two-dimensional array.
twodfft.c (R) - Routine to compute the two-dimensional Discrete Fourier Transform (2-D FFT) of a complex-valued two-dimensional array.
twofft.c (R) - Routine to calculate the first N/2 + 1 components of the Discrete Fourier Transforms (DFT's) of two real-valued data sequences of length N simultaneously.
vitbi.c (R) - Routine to implement a Viterbi decoder.

2.2 Compiling and Linking Applications Programs

Application programs can be compiled and linked as shown below. Be sure that your the model matches with the library model.
The following example uses the "small" model of the Microsoft 'C' 5.0 compiler.

Note: Assume that:

C:\bin has the compiler
C:\lib has the libraries
C:\dspl has the your test source file 'xfft.c'

To compile your source program:

```
c:\dspl\cl /c /AS xfft.c
```

To link the 'xfft.obj' object module with the small model library, spl.lib, to produce the executable module, 'xfft.exe':

```
c:\dspl\link xfft,,,spl
```

You may wish to refer to the Microsoft C manual for more information about compiling and linking files in Microsoft C.

2.3 Error Messages

Errors should not generally be encountered during the use of DSPL. However, some errors due to numerical ill-conditioning may occur as all DSPL routines use single precision floating point numbers (24 bits mantissa and 8 bits exponent). DSPL flags these errors by printing error messages and returning to the system. This feature may be changed to just return to the calling program by changing routine 'splerror' in file 'splutil.c' whose source code is provided with the library, and whose listing appears at the end of this manual. The procedure for making this change is to compile the modified 'splutil.c' and to rebuild the library 'spl.lib' using the microsoft library manager. The following commands accomplish this task:

- cl /c splutil.c
- lib
- .... microsoft screen message should appear...
- Library name 'spl'
- Operations '-splutil +splutil'
- Library cross reference 'spl xrf'
- Output library 'spl'

This procedure saves the old version of the library in 'spl.bak', while the new version will be in 'spl.lib'.

The error message list is given in the Appendix.

3.0 TEST PROGRAMS

3.1 Test Program Overview

Each routine in DSPL (except 'twodfft' and 'twoddct') has been tested using the following test programs. As far as possible, test programs attempt to address the use of DSPL in applications that are likely to arise in practice. Thus, you may, in many instances, be able to use a considerable portion of the software of a test program in your application, or use the software as a guide to program your application.

The input parameters to the test programs are accepted from the keyboard and the output printed on the screen. All screen I/O is alphanumeric except for the output of xorddit.c which writes graphics data. This program would need to be modified slightly to select different video I/O drivers for CGA, EGA, VGA, etc. graphics cards.

3.2 Test Program and Library Cross Reference

The following is a list of test programs provide and the library routines they test:

xbch.c	- elcf.c, encd.c, erlc.c, ffadd.c, ffmul.c, genzec.c, gshf.c, synd.c
xburg.c	- burg.c
xccep.c	- ccep.c, gentwd.c, twofft.c
xchlsky.c	- chlsky.c, cov.c
xconv.c	- acs.c, cencd.c, vitbi.c
xcosft.c	- cosft.c, gentwd.c, icosft.c
xdurbin.c	- acorr.c, durbin.c, genwin.c
xexpdev.c	- expdev.c, ran1.c
xfft.c	- fft.c, gentwd.c
xgasdev.c	- gasdev.c, ran1.c
xlaguer.c	- laguer.c
xmmult.c	- mmult.c
xorddit.c	- orddit.c
xran1.c	- ran1.c
xrbit.c	- rbit.c
xrfft.c	- gentwd.c, irfft.c, rfft.c
xsinft.c	- sinft.c
x2fft.c	- gentwd.c, itwofft.c, twofft.c

3.3 Test Program Listing

The test program source code is provided in Test Program_A and Test Program_B diskettes and program listing is given in the Appendix.

LIBRARY DESCRIPTION

The following is a list of the library modules:

acorr.c (R) - Routine to obtain the autocorrelation sequence of a data record.
acs.c (R) - Routine to implement the add-compare-select logic of the Viterbi Decoder.
burg.c (R) - Routine to obtain the predictor coefficients of a data record directly using the maximum Entropy Method (MEM).
ccep.c (R) - Routine to calculate the complex cepstrum of a data record.
cencd.c (R) - Routine to generate the encoded symbol stream using a convolutional encoder.
chlsky.c (R) - Routine to solve the autocorrelation normal equations using Cholesky decomposition.
cosft.c (R) - Routine to calculate the Discrete Cosine Transform (DCT) of a real-valued data sequence.
cov.c (R) - Routine to obtain the covariance matrix of a data record.
durbin.c (R) - Routine to solve the autocorrelation normal equations using Durbin recursion.
encd.c (R) - Routine to encode an information vector to obtain the code vector.
elcf.c (R) - Routine to find the coefficients of the error locator polynomial.
erlc.c (R) - Routine to find the roots of the error locator polynomial.
expdev.c (S) - Routine to generate an exponentially distributed random variable.
ffadd.c (R) - Routine to add two elements of a finite field.
ffmul.c (R) - Routine to multiply two elements of a finite field.
fft.c (R) - Routine to calculate the Discrete Fourier Transform (DFT) of a complex-valued data record.
gasdev.c (S) - Routine to generate a Gaussian distributed random variable.
gentwd.c (I) - Routine to generate a cosine table that is used by most "transform" routines.
genwin.c (I) - Routine to generate commonly used "windows" for spectral analysis.
genzec.c (I) - Routine to generate the Zech's logarithms of a finite field.
gshf.c (I) - Routine to generate a table of shifted versions of a generator polynomial.
icosft.c (R) - Routine computes the Inverse Discrete Cosine Transform (IDCT) of a real-valued data record.

irfft.c (R) - Routine computes the real valued Inverse Discrete Fourier Transform (IDFT) of length N given its first N/2 + 1 complex-valued Fourier components.
itwofft.c (R) - Routine computes the real-valued Inverse Discrete Fourier Transform (IDFT) of length N given its first N/2 + 1 complex-valued Fourier components.
laguer.c (U) - Given the degree m and the degree m+1 complex coefficients of a polynomial, this routine improves the root of the mth degree polynomial using Laguerre's method.
mmult.c (R) - Routine multiplies N matrices after finding the best order in which to do the multiplications.
orddit.c (R) - Routine to obtain half-tone images from gray-level images using ordered dither.
ran1.c (S) - Routine to generate a uniformly distributed random variable.
rbit.c (S) - Routine to generate a random bit sequence using a maximum length shift register.
rfft.c (R) - Routine to compute the first N/2 + 1 complex components of the Discrete Fourier Transform (DFT) of a real-valued data sequence of length N.
sinft.c - Routine to compute the Discrete Sine Transform (DST) of a real-valued data record.
synd.c (R) - Routine to compute the syndrome from the received vector.
twoddct.c (R) - Routine to compute the two-dimension Discrete Cosine Transform (2-D DCT) of a real-valued two-dimensional array.
twodfft.c (R) - Routine to compute the two-dimensional Discrete Fourier Transform (2-D FFT) of a complex-valued two-dimensional array.
twofft.c (R) - Routine to calculate the first N/2 + 1 components of the Discrete Fourier Transforms (DFT's) of two real-valued data sequences of length N simultaneously.
vitbi.c (R) - Routine to implement a Viterbi decoder.

Module name

acorr.c (R) - Routine to obtain the autocorrelation sequence of a data record

void acorr(float *wsamp, float *ac, int nsamp, int order);

Inputs

wsamp	array of single-precision floating-point numbers representing the windowed input samples.
nsamp	number of samples in the window.
order	number of autocorrelation coefficients to be computed in addition to the energy term, ac[0].

Outputs

ac	array of single precision autocorrelation coefficients, ac[k], k=0,1,...,order.

Processing

This routine is used to generate the kth order autocorrelations, k=0,1,...,order. It uses 'nsamp' products to compute ac[0], 'nsamp'-1 products to compute ac[1], and in general, 'nsamp'-k products to compute ac[k]. In most applications, it is important that the samples have been previously multiplied by a carefully chosen window.

Examples

1) acorr(wsamp,ac,256,4);
This example calculates the autocorrelations ac[0], ac[1], ac[2], ac[3], and ac[4], using a data record that is 256 samples long.
Test program*: xdurbin.c*

Related modules

genwin.c, durbin.c

Module name

acs.c (R) - Routine to implement the add-compare-select logic of the Viterbi Decoder

void acs(float a, float b, float c, float d, float *stamet, long *phist, float *bsm, int *ptr, int skip, int nstates, int *obit);

Inputs

a	top-most branch metric of 'butterfly' (i.e., first state transition when input bit is 0).
b	second branch metric of 'butterfly' (i.e., second state transition when input bit is 0).
c	third branch metric of 'butterfly' (i.e., first state transition when input bit is 1).
d	fourth branch metric of 'butterfly' (i.e., second state transition when input bit is 1).
stamet	floating-point vector of length 'nstates' containing the state metrics of the Viterbi decoder.
phist	long integer vector containing the (8*sizeof(long)) previous bits (the path history) corresponding to each state.
bsm	best (minimum) state metric found so far.
ptr	ptr to first state (in both the 'stamet' and 'phist' arrays).
skip	defines the difference (modulo nstates) between the two points on a 'butterfly'.
nstates	defines the number of states ($2^{(\text{constraint length-1})}$) of the convolutional decoder.

Outputs

stamet	floating-point vector of length 'nstates' containing the updated state metrics after current pass through 'acs'.
phist	long integer vector containing the updated (8*sizeof(long)) previous bits (the path history) corresponding to each of the two states of the butterfly.
bsm	best (minimum) state metric after the current pass through the add-compare-select routine.
obit	the tentative decoded bit for the current pass (this bit corresponds to the symbol received 32+(constraint length-1) bauds before the current baud).

Processing

The add-compare-select routine implements one of the (nstates/2) 'butterflies' in the viterbi decoder. It uses the input branch metrics to update the state metrics and the path history of each 'butterfly' of the Viterbi decoder. It also obtains tentative values for the best state metric and the decoded bit. Last, it updates 'ptr' so that it corresponds to the first state of the next butterfly.

Examples

1) for (i=0; i<nstates; i+=2)
```
acs(mf[perm[i][0]],mf[perm[i+1][0]],mf[perm[i][1]],mf[p
erm[i+1][1]], stamet, phist, bsm, &ptr, *skip, nstates,
obit);
```
This example updates the state metrics in 'stamet' using the branch metrics whose indexes are determined by the array 'perm' (this array is determined by the taps of the convolutional code used). It also updates the path history table 'phist'. The smallest state metric is returned in 'bsm', while the output bit is returned in 'obit'.
Test program: *xconv.c*

Related modules

cencd.c, vitbi.c

Module name

burg.c (R) - Routine to obtain the predictor coefficients of a data record directly using the maximum Entropy Method (MEM)

```
void  burg(float *data, float *pc, float *var, int npts, int order);
```

Inputs

data	vector of single-precision numbers data[i], i=0,1,...,npts-1.
npts	number of samples of input data.
order	order of predictor.

Outputs

pc	vector of single-precision numbers pc[i], i=0,1,...,order, representing the prediction coefficients. It is as assumed that pc[0]=1.0.
var	estimate of the variance of the driving noise of the AR process.

Processing

This routines uses the data samples directly to estimate the predictor coefficients using the Maximum Entropy Method (MEM).

Examples

1) burg(samp,pc,var,260,4);
This example calculates the optimum fourth order predictor over the given 260 samples in the array "samp" using the Burg iteration technique. The driving noise variance estimate is returned in var, while the estimated predictor coefficients are stored in the array pc in locations whose indexes are 1, 2, 3, and 4.
Test program: *xburg.c*

Related modules

durbin.c, chlsky.c

Module name

ccep.c (R) - Routine to calculate the complex cepstrum of a data record

void ccep(float *datar, float *twiddle, float *logmag, float *conpha, float thd1, float thd2, int N);

Inputs

datar	array of single-precision floating-point numbers constituting the real valued input data sequence.
twiddle	array of length 5N/4 of single-precision floating-point numbers containing the quantities $\cos(2\pi k/N)$, k = 0,1,...,(5N/4)-1.
thd1	value of the threshold used to check the consistency of the phase with reference to the value of estimated linear phase increment.
thd2	value of threshold used to check the consistency of the phase with reference to the estimated phase value.
N	length of the number of points in the input sequence, where N is a power of 2.

Outputs

logmag	array of length N/2 + 1 containing the first N/2 + 1 log-magnitude samples of the real valued input data sequence.
conpha	array of length N/2 + 1 containing the first N/2 + 1 continuous phase samples of the real valued input data sequence.

Processing

This routine computes the complex cepstrum samples of a real valued data record of length N, where N is a power of 2. The first part of the routine computes the DFT of the input sequence x[n] and nx[n] using the library function "twofft". The need to compute the DFT of nx(n) stems from the fact that the derivative d/dw[X(jw)] = -j*DFT of nx(n) is needed to compute the continous phase. Next, the log of the magnitude, principal phase value, and the phase derivative at each $w_k = 2\pi k/N$, where k = 0,1...,N/2 is computed. At each w_k, a phase estimate is initially formed by one-step trapezoidal integration, starting at w_{k-1}. If the resultant estimate is not consistent, the adaptive integration scheme is applied within the interval $[w_{k-1}, w_k]$. The step size of the adaptation is carefully designed to minimize the number of extra DFT's required. The search for consistency is done by consecutively splitting the step interval in half. Recursive programming is employed to make the parameters available at each intermediate frequency for computation of the DFT, the principal phase, and the phase derivative. Finally, the linear phase component is removed from the unwrapped phase. The vectors logmag and the conpha contain the log magnitude and the continuous phase values of first N/2 + 1 samples of real valued data record of length N.

Examples

1) gentwd (twiddle, 32);
    ccep(datar,twiddle,logmag, conpha, 0.8π,0.5π,32);

This computes the 17 cepstrum samples for the real valued record of length 32. The thresholds 0.8π and 0.5π are application dependent (reference: Jose M. Tribolet, vol. ASSP-25, No.2, April 1977. "A New Phase Unwrapping Algorithm"). It is recommended that N should be more than the actual data length (typically 4 times) to minimize the number of extra DFT computations required in the adaptive integration scheme.
Test program: *xccep.c*

Related modules

gentwd.c, twofft.c.

Module name

cencd.c (R) - Routine to generate the encoded symbol stream using a convolutional encoder

unsigned int cencd (int bit, unsigned int *state, int rate, unsigned int *taps)

Inputs

bit	input bit to the convolutional encoder.
state	the state of the convolutional encoder (only the k-1 right-most bits are relevant for a constraint length k code).
rate	the inverse of the rate of the code.
taps	vector containing 'rate' tap masks that define the convolutional code.

Outputs

state	the updated state of the convolutional encoder (consisting of the old state shifted up by one bit, with the input bit occupying the LSB).
cencd	the symbol output by the convolutional encoder.

Processing

The convolutional encoder first shifts up 'state' by one bit and inserts the input bit into the LSB of 'state'. It then obtains the parity of 'state' masked by the different taps. Finally, it packs the 'rate' different parities obtained into 'cencd' to represent the output symbol.

Examples

1) symb=cencd(bit,&state,2,taps);
This example updates the state of a rate 1/2 convolutional encoder, defined by the vector 'taps', as a function of the input 'bit' and the old 'state'. The output in this case is a 4-ary symbol.
Test program: *xconv.c*

Related modules

acs.c, vitbi.c

Module name

chlsky.c (R) - Routine to solve the autocorrelation normal equations using Cholesky decomposition.

void chlsky(float **cc, int order, float *pc);

Inputs

cc	matrix of single-precision numbers cc[i][j], i, j=0,1,...,order.
order	dimension of N x N covariance matrix N=order+1.

Outputs

pc	vector of single-precision numbers pc[i], i=0,1,...,order-1, representing the prediction coefficients.

Processing

This routine uses the (order+1) x (order+1) covariance matrix computed by cov.c to obtain the prediction coefficients. Note that cc[0][0] is not used by the routine. The method used is Cholesky decomposition where the positive definite symmetric submatrix v[i][j], i,j=0,1,..,order-1=cc[i][j], i,j=1,2,...,order, is decomposed into the product of a lower triangular matrix, a diagonal matrix, and a upper triangular matrix (which is the transpose of the lower triangular matrix). The system of linear equations is v[i][j]*pc[i]=cc[i+1][0], i,j=0,1,...,order-1. The predictor coefficients, pc[i], i=0,1,...,order-1, are then computed by back-substitution.

Examples

1) cov(samp,cc,260,4);
   chlsky(cc,4,pc);

This example calculates the covariance matrix cc[i][j], i,j=0,1,2,3,4 using a data record that is 260 samples long. The set of four linear equations implied by the covariance matrix are then solved by Cholesky decomposition to yield the predictor coefficients.
Test program: *xchlsky.c*

Related modules

cov.c

Module name

cosft.c (R) - Routine to calculate the Discrete Cosine Transform (DCT) of a real-valued data sequence.

void cosft(float *twiddle, float *data, int N);

Inputs

twiddle	array of length 5N/2 of single-precision floating-point numbers containing the quantities $\cos(2\pi k/(2N))$, $k=0,1,\ldots,(5N/2)-1$.
data	array of single-precision floating-point numbers constituting the real-valued input data sequence.
N	length of the number of points in the input sequence, where N is a power of 2.

Outputs

data	array of length N constituting the components of the cosine transform of the real-valued input data.

Processing

This routine computes the cosine transform of a real input sequence, x[n]. The first part of the routine forms the sequence,
$y[n]=(x[n]+x[N-n])/2-(x[n]-x[N-n])\sin(\pi n/N)$, $n=1,2,\ldots,N-1$,
$y[0]=x[0]$,
and computes the DFT, $Y[k]$, $k=0,1,\ldots,(N/2)-1$, using a routine identical to 'rfft' except for the indexing into the twiddle factor array. The second part of the routine computes the cosine transform of x[n], C(m), $m=1,\ldots,N-1$, from Y[k], $k=0,1,\ldots,(N/2)-1$ as follows:
$C[2k+1]=C[2k-1]-\mathrm{Im}\{Y[k]\}$, $k=1,2,\ldots,(N/2)-1$, where the starting point of the recursion is
$C[1]= \Sigma_m x[m]\cos(m\pi/N)$, $m=0,1,\ldots,N-1$, and
$C[2k]=\mathrm{Re}\{Y[k]\}$, $k=0,1,2,\ldots,(N/2)-1$.

Examples

1) gentwd (twiddle,64);
   cosft (twiddle,data,32);
This computes the cosine transform of a 32-point real-valued sequence. The output is computed in place.
Test program: *xcosft.c*

Related modules

gentwd.c, fft.c, rfft.c, sinft.c, icosft.c

Note

Unfortunately, no one standard definition applies to the "Cosine Transform". This implementation supplies $C[k] = \Sigma_n x[n]\cos(\pi nk/N)$. This is the desired form when transform methods are used to solve differential equations where the derivatives of the solutions are zero at the boundaries.
The differences between the definitions arise from different samplings of the kernel in the original and in the transform domain. The form that is most suited to image processing applications defines C[k] as $\Sigma_n \surd(2/n)x[n]\cos((n+1/2)k\pi/N)$ for $k=1,2,\ldots,N-1$, and for the zeroth component $\surd(1/N)$ times the value given by 'cosft' for the zeroth component. Note that this can be obtained (except for a scale factor) by first calling 'cosft' and then multiplying each component i of the transform by $\cos((\pi i)/(2N))$. This form is implemented in the 2-D cosine transform routine of DSPL.

Module name

COV.C (R) - Routine to obtain the covariance matrix of a data record.

void	cov(float *samp, float **cc, int buflen, int order);

Inputs

samp	array of single-precision floating-point numbers representing the input samples.
buflen	number of samples in the buffer must be equal to the number of samples used in the averaging plus a number of samples equal to 'order'.
order	dimension of N x N covariance matrix N=order+1.

Outputs

cc	matrix of single-precision numbers cc[i][j], i,j=0,1,...,order.

Processing

This routine is used to generate the (order+1) x (order+1) covariance matrix using buflen-order products to compute each element of the matrix. The matrix is symmetric, but not Toeplitz.

Examples

1) cov(samp,cc,260,4);
This example calculates the covariance matrix cc[i][j], i,j=0,1,2,3,4 using a data record that is 260 samples long. The sum of 256 products is used to compute each element of the matrix.
Test program: *xchlsky.c*

Related modules

chlsky.c

Module name

durbin.c (R) - Routine to solve the autocorrelation normal equations using Durbin recursion.

void durbin(float *ac, int order, float *a, float *k, float *e);

Inputs

ac	vector of single-precision numbers ac[i] i=0,1,...,order, representing the autocorrelation sequence.
order	order of the the predictor.

Outputs

a	vector of single-precision numbers a[i], i=0,1,...,order, representing the prediction coefficients.
k	vector of single-precision numbers k[i], i=1,...,order-1, representing the reflection coefficients.
e	vector of single-precision numbers e[i], i=0,1,...,order-1, representing the ratio of the power of the residual signal to the power in the input signal (ac[0]).

Processing

This routine uses the (order+1) long autocorrelation sequence computed by acorr.c to obtain the prediction coefficients, the reflection coefficients, and the error sequence. The method used is Durbin recursion, where each successively higher model is built recursively from the previous model. The reflection coefficients and the (normalized) variance sequence are by-products of this procedure.

Examples

```
1) acorr(samp,ac,260,4);
   durbin(ac,4,pc,rc,e);
```

This example calculates the autocorrelation sequence ac[i], i=0,1,2,3,4 using a data record that is 260 samples long. The set of four linear equations implied by the autocorrelation sequence are then solved by Durbin's recursion to yield the prediction coefficients, pc, the reflection coefficients, rc, and the error variance sequence, e. Test program: *xdurbin.c*

Related modules

acorr.c

Module name

encd.c (R) - Routine to encode an information vector to obtain the code vector.

void encd(int *inf, int *shfg, int blkl, int genl, int ilen);

Inputs

inf	vector containing the left-justified packed information polynomial (8*sizeof(int) bits to a word).
shfg	vector containing all possible shifted versions of the generator polynomial.
blkl	length of the codeword in bits.
genl	length of the generator polynomial in words.
ilen	length of the information polynomial in bits.

Outputs

inf	vector containing the check bits appended to the original information polynomial, packed (8*sizeof(int)) bits to the word and left-justified.

Processing

This routine uses all possible shifted versions of the generator polynomial in the array 'shfg' to divide x^k*inf(x) by the generator polynomial and thereby obtain the check polynomial. The check polynomial is then appended to the original information polynomial to yield the (systematic) codeword.

Examples

1) encd(inf,shfg,127,3,99);
This example uses the shifted 8*sizeof(int) versions of the generator polynomial to divide x^{28}*inf(x) by the generator polynomial for the (127,99) BCH code. It then appends the 28 check bits thus obtained to the original information polynomial inf(x).
Test program: *xbch.c*

Related modules

gshf.c

Module name

elcf.c (R) - Routine to find the coefficients of the error locator polynomial.

```
void elcf(int *synv, int *zech, int *cdv, int blen, int cerc, int *nerr);
```

Inputs

synv	vector of integers, 2*cerc long, containing the syndrome of the received codeword.
zech	array of integers indexed by i=0,1,...,$2^{(m-1)}$, containing the Zech's logarithm values defined by $\alpha^{zech[i]}$ = XOR(1,α^i).
blen	number of bits in the block.
cerc	number of errors correctable by the code.

Outputs

cdv	vector of length nerr+1 containing the exponents of the coefficients of the error locator polynomial.
nerr	number of errors estimated by 'elcf'.

Processing

The coefficients of the error locator polynomial are calculated from the syndrome using the Massey-Berlekamp algorithm.

Examples

1) elcf(synv,zech,cdv,127,4,&nerr);

This example calculates the error locator polynomial from the syndrome of rcv[x] in the vector 'cdv' (length=5) for the 4-error correcting (127,99) BCH code.
Test program: *xbch.c*

Related modules

ffmul.c, ffadd.c, synd.c, erlc.c

Module name

erlc.c (R) - Routine to find the roots of the error locator polynomial

void erlc(int, *cdv, int *zech, int *loc, int blen, int nerr);

Inputs

cdv	vector of length nerr+1 containing the exponents of the coefficients of the error locator polynomial.
zech	array of integers indexed by i=0,1,...,$2^{(m-1)}$, containing the Zech's logarithm values defined by $\alpha^{zech[i]}$ = XOR(1,α^i).
blen	number of bits in the block.
nerr	number of errors estimated by 'elcf'.

Outputs

loc	vector of length 'nerr' containing the error locations (if the decode was successful).
erlc	erlc's value is 1 for a successful decode and 0 otherwise.

Processing

The roots of the error locator polynomial are determined by evaluating $x^{(nerr)}+c[1]x^{(nerr-1)}+...+c[nerr]$ at $x=\alpha^i$, i=0,1,...,blen-1. If the number of roots thus found is equal to the number of errors estimated by 'elcf', 'erlc' returns a value of 1 (to indicate a successful decode) and 0 otherwise.

Examples

1) succ=erlc(cdv,zech,loc,127,nerr);
This example calculates the roots of the error locator polynomial for the 4-error correcting (127,99) BCH code and returns a value of 1 in 'succ' if the decode was successful
Test program: *xbch.c*

Related modules

ffmul.c, ffadd.c, elcf.c

Module name

expdev.c (S) - Routine to generate an exponentially distributed random variable.

float expdev(int *idum);

Inputs

idum	Initializing variable relayed to ran1 routine. If this variable is negative, the routine 'ran1' re-initializes itself. Upon initialization, the routine 'ran1' returns a value of of idum = +1. Thus, on subsequent calls, the user need not specifically set this input.

Outputs

expdev	exponentially distributed positive floating-point number with mean=1.

Processing

This routine is used to generate exponentially distributed random numbers. If the argument 'idum' is negative, the random number generator is reinitialized; otherwise, a new random number is generated. The routine first calls 'ran1' to obtain a uniformly distributed random number, x, between 0 and 1. It then uses the transformation y=-ln(x) to obtain an exponentially distributed random number with mean 1.0.

Examples

```
1) idum=-1;
   expdev(&idum);
```
This example initializes the random number generator and returns the first random number.

```
2) expdev(&idum);
```
This example is used on subsequent calls to expdev.
Test program: *xexpdev.c*

Related modules

ran1.c

Module name

ffadd.c (R) - Routine to add two elements of a finite field.

int ffadd(int *zech, int a, int b, int flen);

Inputs

zech	array of integers indexed by $i=0,1,...,2^{(m-1)}$, containing the Zech's logarithm values defined by $\alpha^{zech[i]} = XOR(1,\alpha^i)$.
a	exponent of first finite field element, where $0 \le a < flen$, or $a=-\infty$ (represented by -1).
b	exponent of second finite field element, where $0 \le b < flen$, or $b=-\infty$ (represented by -1).
flen	number of elements in the finite field.

Outputs

ffadd	result of addition represented by $\alpha^{ffadd} = \alpha^a + \alpha^b$.

Processing

This routine adds two finite field elements by returing the larger of the two exponents if the smaller of the two exponents is less than zero, or by returning the result of routine 'ffmul' with input exponents equal to the smaller of the two exponents and zech[larger exponent-smaller exponent].

Examples

1) ffadd(zech,28,37,127);
This example calculates $\alpha^{ffmul} = \alpha^{28} + \alpha^{37}$ in a finite field of 127 elements.
Test program: *xbch.c*

Related modules

ffmul.c

Module name

ffmul.c (R) - Routine to multiply two elements of a finite field.

int ffmul(int a, int b, int flen);

Inputs

a	exponent of first finite field element, where $0 \le a < \text{flen}$, or $a = -\infty$ (represented by -1).
b	exponent of second finite field element, where $0 \le b < \text{flen}$, or $b = -\infty$ (represented by -1).
flen	number of elements in the finite field.

Outputs

ffmul	result of multiply represented by $\alpha^{\text{ffmul}} = \alpha^{a} * \alpha^{b} = \alpha^{(a+b)\text{mod flen}}$, when neither a nor b is $-\infty$ (represented by -1). When either a or b is -1, then the result is -1.

Processing

This routine multiplies two finite field elements by adding their exponents modulo the finite field length, when neither a nor b is $-\infty$. When either a or b is $-\infty$, then the result is $-\infty$.

Examples

1) ffmul(28,37,127);
This example calculates $\alpha^{\text{ffmul}} = \alpha^{28} * \alpha^{37}$ in a finite field of 127 elements.
Test program: *xbch.c*

Related modules

ffadd.c

Module name

fft.c (R) - Routine to calculate the Discrete Fourier Transform (DFT) of a complex-valued data record.

void fft(float *twiddle, float *datar, float *datai, int N);

Inputs

twiddle	array of length 5N/4 of single-precision floating-point numbers containing the quantities $\cos(2\pi k/N)$, $k=0,1,...,(5N/4)-1$.
datar	array of single-precision floating-point numbers constituting the real part of input data.
datai	array of single-precision floating-point numbers constituting the imaginary part of input data.
N	length of the DFT (i.e., number of complex inputs/outputs), where N is a power 2.

Outputs

datar	array of single-precision floating-point numbers constituting the real part of the DFT.
datai	array of single-precision floating-point numbers constituting the imaginary part of the DFT.

Processing

This routine does an in-place DFT of a complex input sequence, i.e., the result for each stage is stored in the same array as the previous stage.The first part of the routine re-arranges the input in bit-reversed order. The remainder of routine consists of three nested loops: The first loop does lg(N) (logarithm to the base 2 of N) iterations corresponding to the number of stages in the FFT algorithm. The second loop changes the index into the twiddle factor array as required. The innermost loop does the butterfly computations for which the twiddle factor indices remain constant.

Examples

```
1) gentwd (twiddle,32);
   fft (twiddle,datar,datai,32);
```

This computes a 32-point DFT and stores the result in datar and datai. 'twiddle' is the array of twiddle factors which are calculated using routine 'gentwd'.

```
2) fft(twiddle,datai,datar,N);
```

The inverse DFT of a complex sequence can be computed (to within a scale factor) by calling the fft routine with the real and imaginary data reversed in order. If the exact inverse is desired, then each of the components of datai and datar should be multiplied by 1/N.
Test program: *xfft.c*

Related modules

gentwd.c, rfft.c, irfft.c, twofft.c, itwofft.c, ccep.c, cosft.c, sinft.c

Module name

gasdev.c (S) - Routine to generate a Gaussian distributed random variable.

float gasdev(int *idum);

Inputs

idum	initializing variable relayed to ran1 routine. If this variable is negative, the routine 'ran1' re-initializes itself. Upon initialization, the routine 'ran1' returns a value of of idum = +1. Thus, on subsequent calls, the user need not specifically set this input.

Outputs

gasdev	Gaussian distributed floating-point number with mean=0 and variance=1.

Processing

This routine is used to generate Gaussian distributed random numbers. If the argument 'idum' is negative, the random number generator is reinitialized; otherwise, a new random number is generated. On every alternate call, the routine makes two calls 'ran1' to obtain two uniformly distributed random numbers, v1 and v2, between 0 and 1. It then uses the Box-Muller transformation to obtain two Gaussian distributed random numbers with zero mean and unit variance. It outputs one of these and saves the other for the next call to 'gasdev'.

Examples

```
1) idum=-1;
   gasdev(&idum);
```
This example initializes the random number generator and returns the first random number.

```
2) gasdev(&idum);
```
This example is used on subsequent calls to gasdev.
Test program: *xgasdev.c*

Related modules

ran1.c

Module name

gentwd.c (I) - Routine to generate a cosine table that is used by most "transform" routines.

void gentwd(float *twiddle, int N);

Inputs

N	number of points to be sampled on the unit circle, N being a power of 2.

Outputs

twiddle	array of single-precision floating-point numbers of length 5N/4 containing the quantities $\cos(2\pi k/N)$, k = 0,1,...,(5N/4)-1. Thus, the cosine table begins at location twiddle[0], while the sine table index begins at twiddle[N/4].

Processing

This routine iteratively calculates the quantities $\cos(2\pi k/N)$, k=0,1,...,(5N/4)-1.

Examples

1) gentwd(twiddle,32);
This example generates twiddle factors for a 32 point DFT and stores them in the array 'twiddle' of length (5*32)/4.
Test program: *xfft.c* (and 'transform' routines)

Related modules

fft.c, rfft.c, irfft.c, twofft.c, itwofft.c, cosft.c, sinft.c, ccep.c

Module name

genwin.c (I) - Routine to generate commonly used "windows" for spectral analysis.

void	genwin(float *win, float scale, float parm, int wtyp, int n);

Inputs

scale	scale factor to be incorporated into the window function
type	integer that specifies the type of the window. The different options available are: 0 - triangular or Bartlet window 1 - cosine^x window (parameter required) 2 - Hamming window 3 - Blackman window 4 - Blackman-Harris window 5 - Tukey window (parameter required) 6 - Poisson window (parameter required) 7 - Hanning-Poisson window (parameter required) 8 - Cauchy window (parameter required) 9 - Gaussian window (parameter required) 10 - Dolph-Chebyshev window (parameter required) 11 - Kaiser-Bessel window (parameter required) The names TRIANGLE, COSPOWER, HAMMING, BLACKMAN, BLACKMAN_HARRIS, TUKEY, POISSON, HANN_POISSON, CAUCHY, GAUSSIAN, DOLPH_CHEBYSHEV, and KAISER_BESSEL may be used in place of the numbers 0,1,...,11 if the file "genwin.h" is included in the driver routine
parm	parameter for the window (if required). If a parameter is not required, the routine ignores this argument.
N	number of points in the window

Outputs

win	array of single-precision floating-point numbers of length N containing the samples of the window.

Processing

This routine calculates the samples of the window specified by the "type" input. The window functions implemented are given below (omitting the scale factor):

For a triangular window:

```
                { 2n/N, n=0,1,...,N/2
win[n] =        {
                { win[N-n], n=N/2,...,N-1
```

For a cosine^x window:
win[n] = $(\sin(n\pi/N))^x$, n=0,1,...,N-1 (x=2 is the well-known Hanning window)

For a Hamming window:
win[n] = 0.54 - 0.46 cos(2πn/N), n=0,1,...,N-1

For a Blackman window:
win[n] = 0.42-0.5cos(2πn/N)+0.08cos(4πn/N), n=0,1,...,N-1

For a Blackman-Harris window:
win[n] = 0.35875-0.48829cos(2πn/N)+0.14128cos(4πn/N)-0.01168cos(6πn/N), n=0,1,...,N-1

For a Tukey window:

```
                {1.0, 0≤|n-N/2|≤parm*N/2,
                {       n=0,1,...,N-1
win[n] =        {
                {0.5*{1+cos(π(n(1+parm)*N/2)/
                { ((1-parm)*N))}, n=0,1,...,N-1
```

For a Poisson window:
win[n] = exp(-2*parm*|n-N/2|/N), n=0,1,...,N-1

For a Hanning-Poisson window:
win[n] = the product of the Hanning and Poisson windows.

For a Cauchy window:
win[n] = $1/\{1+(2.0*parm*(n-N/2)/N)^2\}$,   n=0,1,...,N-1

For a Gaussian window:
win[n] = $\exp\{-0.5(2.0*parm*(n-N/2)/N)^2\}$,   n=0,1,...,N-1

For a Dolph-Chebyshev window:
win[n] = DFT{WIN[k]}, n,k=0,1,...,N-1, where
WIN[k] = $(-1)^k$cos(N*arccos(b*cos(πk/N)))/
cosh(N*arccosh(b)), k=0,1,...,N-1, where
b=cosh(arccosh(10^{parm})/N), and
arccos(x) is defined for all real x as follows:

$$\arccos(x) = \begin{cases} \pi/2-\tan^{-1}(x/\sqrt{(1-x^2)}), & |x| < 1.0 \\ \\ \text{arccosh}(x), & |x| \geq 1.0 \end{cases}$$

For a Kaiser-Bessel window:
$win[n] = I_0(\pi * parm * \sqrt{(4n/N(1-n/N))}/I_0(\pi * parm)$, $n=0,1,\ldots,N-1$,
where I_0 is the modified zeroth order Bessel function of the first kind.

Examples

1) genwin(win,1.0,parm,HAMMING,32);
This example generates the samples of a 32 point Hamming window and stores them in the array "win". The window is scaled by unity.
Test program: *xdurbin.c*

Related modules

acorr.c

Module name

genzec.c (I) - Routine to generate the Zech's logarithms of a finite field

void genzec(int m, int p, int *zech);

Inputs

m	exponent of the binary Galois field GF(2^m). The maximum and minimum values of m permitted are (8*sizeof(int)-1) and 2 respectively.
p	bit pattern defining the primitive element, α, of GF(2^m). If p is represented as $p(m-1)*2^{m-1}+p(m-2)*2^{m-2}+\ldots+p(1)*2+p(0)$, then the primitive element, α, is defined by $\alpha^m = p(m-1)*\alpha^{m-1}+p(m-2)*\alpha^{m-2}+\ldots+p(0)$.

Outputs

zech	array of integers indexed by $i=0,1,\ldots,2^{m-1}$, containing the Zech's logarithm values defined by $\alpha^{zech[i]}=$ ffadd(1,α^i).

Processing

This routine first calculates the finite field exponent values in the table 'zech'. Next it inserts the index of j in location i when XOR(zech[i],1)=zech[j], and vice versa. This routine is used by the finite field addition routine.

Examples

1) genzec(7,9,zech);
This example generates the Zech's logarithms for GF(2^7) where the primitive element, α, is defined by $\alpha^7=\alpha^3+1$.
Test program: *xbch.c*

Related modules

ffadd.c

Module name

gshf.c (I) - Routine to generate a table of shifted versions of a generator polynomial.

void gshf(int *shfg, int *genp, int genl);

Inputs

genp	generator polynomial of the binary code. This polynomial is packed into 2+(k-1)/(8*sizeof(int)) words, where k is the number of check bits in the code. The most significant bit of genp[0] represents the coefficient of x^k (which is always 1).
genl	length of the generator polynomial in words. This length is given by 2+(k-1)/(8*sizeof(int)).

Outputs

shfg	array of right shifted versions of the generator polynomial for shift values=0,1,...,8*sizeof(int)-1.

Processing

This routine stores all possible shifted versions of the generator polynomial in the array 'shfg'. This expedites the long division process used to encode the information polynomial.

Examples

```
1) unsigned int genp[3]={0xe4e1,0x35c8,0x0000};
   unsigned int *shfg;
   shfg=uvector(0,(24*sizeof(int)-1);
   gshf(shfg,genp,3);
```

This example generates the shifted versions of the generator polynomial for the (127,99) BCH code.
Test program: *xbch.c*

Related modules

encd.c

Module name

icosft.c (R) - Routine computes the Inverse Discrete Cosine Transform (IDCT) of a real-valued data record.

void icosft(float *twiddle, float *data, int N);

Inputs

twiddle	array of length 5N/2 of single-precision floating-point numbers containing the quantities $\cos(2\pi k/(2N))$, $k=0,1,\ldots,(5N/2)-1$.
data	array of single-precision floating-point numbers constituting the real-valued input data sequence, where N is a power of 2.
N	number of single-precision floating-point number in the input sequence, where N is a power of 2.

Outputs

data	array of length N constituting the components of the inverse cosine transform of the real-valued input data.

Processing

This routine computes the inverse cosine transform of a real input sequence, x[n]. The first part of the routine takes the cosine transform of the input sequence to obtain C[n]. The second part of the routine computes the desired inverse cosine transform (except for a scale factor) of x[n], C'[n], from C[n] in place as follows:

$A = \Sigma_{\text{even\_terms}}$ C[n], $B = \Sigma_{\text{odd\_terms}}$ C[n],
sumo = $\Sigma_{\text{odd\_terms}}$ of C'[n] = C[0]-2(A-B),
sume = $\Sigma_{\text{even\_terms}}$ of C'[n] = (2B)/N-sumo,
C'[0] = A-BC'[2k+1] = C[2k+1]-sume, k=0,1,...,(N/2)-1,
C'[2k] = C[2k]-sumo, k=1,2,...,(N/2)-1.
If the exact inverse cosine transform is desired, the output array should be multiplied by 2/N.

Examples

1) gentwd (twiddle,64);
   icosft (twiddle,data,32);

This computes the inverse cosine transform of a 32-point real-valued sequence. The output is computed in place. If the exact inverse is desired, the output array should be scaled by 1/16.
Test program: *xcosft.c*

Related modules

gentwd.c, fft.c, rfft.c, sinft.c, cosft.c

Module name

irfft.c (R) - Routine computes the real valued Inverse Discrete Fourier Transform (IDFT) of length N given its first N/2+1 complex-valued Fourier components.

void irfft(float *twiddle, float *data, int N);

Inputs

twiddle	array of length 5N/4 of single-precision floating-point numbers containing the quantities cos($2\pi k/N$), k = 0,1,...,(5N/4)-1.

data	array of length N (2 real and (N-1)/2 complex single-precision floating-point numbers) constituting the first (N/2)+1 components of the DFT of the real-valued output data. The components should be stored in the array 'data' in the following order:
	data[0]=Re{X[0]} (note that Im{X[0]}=0)
	data[1]=Re{X[N/2]} (note that Im{X[N/2]}=0)
	data[2m]=Re{X[m]}, m=1,2,...,(N/2)-1
	data[2m+1]=Im{X[m]}, m=1,2,...,(N/2)-1.

N	number of single-precision floating-point numbers in the input sequence, where N is a power of 2.

Outputs

data	array of single-precision floating-point numbers constituting the real-valued input data sequence.

Processing

This routine computes the inverse DFT (to within a scale factor of N/2) of an even conjugate symmetric sequence. The first part of the routine unscrambles the input obtain a N/2 point complex sequence, Z[k], k=0,1,...,(N/2)-1, whose DFT is the sequence x[2m]-j*x[2m+1], m=0,1,...,(N/2)-1, where x[n], n=0,1,...,N-1, is the desired output. Thus, the remaining part of the routine takes the DFT of Z[k] and negates every imaginary component of the DFT to obtain x[n].

Examples

1) gentwd (twiddle,32);
   irfft (twiddle,data,32);

This computes the inverse DFT of an even symmetric sequence to obtain a 32-point real-valued sequence. Although the input data array contains only 17 Fourier components, 2 of them real and 15 of them complex, the remaining 15 complex components are implied by: X[32-k] = conj{X[k]}, k=1,2,...,15, i.e., Re{X[32-k]} = data[2k], k=1,2,...,15, and Im{X[32-k]} = data[2k+1], k=1,2,...,15. If the exact inverse DFT is desired, then each of the components of 'data' should be multiplied by 2/N.
Test program: *xrfft.c*

Related modules

gentwd.c, fft.c, rfft.c, twofft.c, itwofft.c

Module name

itwofft.c (R) - Routine computes the real-valued Inverse Discrete Fourier Transform (IDFT) of length N given its first N/2+1 complex-valued Fourier components.

void itwofft(float *twiddle, float *data1, float *data2, N);

Inputs

twiddle	array of length 5N/4 of single-precision floating-point numbers containing the quantities cos(2πk/N), k = 0,1,...,(5N/4)-1.

data1	array of length N (2 real and (N-1)/2 complex single-precision floating-point numbers) constituting the first (N/2)+1 components of the DFT of the first real-valued input data. The symmetry property X[k] = conjugate of X[N-k] is assumed for the remaining (N/2)-1 Fourier components. The components should be stored in the array 'data1' in the following order:
	data1[0]=Re{X[0]} (note that Im{X[0]}=0)
	data1[N/2]=Re{X[N/2]} (note that Im{X[N/2]}=0)
	data1[m]=Re{X[m]}, m=1,2,...,(N/2)-1
	data1[N-m]=Im{X[m]}, m=1,2,...,(N/2)-1.

data2	array of length N (2 real and (N-1)/2 complex single-precision floating-point numbers) constituting the first (N/2)+1 components of the DFT of the second real-valued input data. The symmetry property Y[k] = conjugate of Y[N-k] is assumed for the remaining (N/2)-1 Fourier components. The components should be stored in the array 'data2' in the following order:
	data2[0]=Re{Y[0]} (note that Im{Y[0]}=0)
	data2[N/2]=Re{Y[N/2]} (note that Im{Y[N/2]}=0),
	data2[m]=Re{Y[m]}, m=1,2,...,(N/2)-1,
	data2[N-m]=Im{Y[m]}, m=1,2,...,(N/2)-1.

N	length of the number of points in the input sequences, where N is a power of 2.

Outputs

data1	array of single-precision floating-point numbers constituting the first real-valued output data sequence.
data2	array of single-precision floating-point numbers constituting the second real-valued output data sequence.

Processing

This routine computes the inverse DFT's of two complex input sequences simultaneously. The first part of the routine computes Z[k], k=0,1,...,N/2, from X[k] and Y[k], k=0,1,...,N-1. Since the dc (0th) and the N/2th frequency components of the DFT's of the real valued sequences are real, and the remaining (N/2)-1 components complex, N locations are adequate to store the input DFT sequences of x[n] and y[n] in an invertible form. The second part of the routine computes the complex sequence, z[n]=x[n]+jy[n], n=0,1,...,N-1, by computing an in-place DFT of Z[k], k=0,1,...,N-1.

Examples

1) gentwd (twiddle,32);
   itwofft (twiddle,data1,data2,32);

This computes the inverse DFT of two sequences simultaneously. Each input sequence contains only 17 Fourier components, 2 of them real and 15 of them complex. The remaining 15 complex components are assumed as follows:
X[32-k] = conj{X[k]}, k=1,2,...,15,
i.e., Re{X[32-k]} = data1[k], k=1,2,...,15, and
Im{X[32-k]} = -data1[32-k], k=1,2,...,15, and
Y[32-k] = conj{Y[k]}, k=1,2,...,15,
i.e., Re{Y[32-k]} = data2[k], k=1,2,...,15, and
Im{Y[32-k]} = -data2[32-k], k=1,2,...,15.
The two sequences x[n] and y[n] are stored in data1 and data2 respectively
Test program: *x2fft.c*

Related modules

gentwd.c, fft.c, rfft.c, irfft.c, twofft.c

Module name

laguer.c (U) - Given the degree m and the degree m+1 complex coefficients of a polynomial, this routine improves the root of the mth degree polynomial using Laguerre's method.

void laguer(fcomplex *data, int M, fcomplex *x, float eps, int polish);

Inputs

data	array of (degree+1) structures of type complex constituting the complex coefficients of the polynomial, where the real and imaginary parts are single precision floating-point numbers. The structure data[0] corresponds to the constant term and the structure data[degree] corresponds to the coefficient of the highest power of x.
M	integer value specifying the degree of the polynomial.
x	pointer to the complex number constituting trial value.
eps	single precision floating-point number specifying the desired fractional accuracy. This parameter is relevant only when 'polish' is input as zero.
polish	an input integer value; when it is non-zero, the routine ignore eps and instead attempts to improve x (assumed to be a good initial guess) to the achievable roundoff limit.

Outputs

x	pointer to the structure constituting one of the roots of the given polynomial.

Processing

This routine evaluates the root of the polynomial whose coefficients can be complex, using Laguerre's method. The vector data contains the degree+1 structs of type complex corresponding to the coefficients of the polynomial. The struct data[0] contains the value of the constant term where as data[degree] contains the complex coefficient of the highest power of x. The function proceeds on the basis of a trial root, and attempts to converge to a true root. The root to which it converges depends on the trial value. The method operates iteratively as follows: for a trial value x, an intermediate variable 'a' which signifies the distance between the trial value x and the actual root, is estimated. Then x-a becomes the next trial value.

This process continues until 'a' is sufficiently small or until the roundoff limit is encountered

Examples

1) laguer(data,20,*x,1.0e-6,0);
This example finds a root of a polynomial of degree 20 whose complex coefficients are stored in the vector 'data'. The trial value x is improved using Laguerre's method until the desired fractional accuracy (1.0e-6 in this case) is achieved.

2) laguer(data,20,*x,1.0e-6,1);
This example finds a root of a polynomial of degree 20 whose complex coefficients are stored in the vector 'data'. The trial value x is improved using Laguerre's method until the roundoff limit is reached. The desired fractional accuracy is ignored in this case.
Test program: *xlaguer.c*

Related modules
none.

Module name

mmult.c (R) - Routine multiplies N matrices after finding the best order in which to do the multiplications.

mptr mmult(mtx *data, int N);

Inputs

data	array of structures corresponding to the number of matrices to be multiplied. Each structure contains a pointer to the matrix, dimension of the matrix, an integer flag value which must be initialized to 1 before invoking the function.
N	integer value specifying the number of matrices to be multiplied.

Outputs

mmult	pointer to a structure which contains a pointer and dimension of the the resultant matrix.

Processing

This routine optimizes the order of evaluation of the product of N matrices by using dynamic programming. Let m(i,j) be the minimum cost of computing M[i]*M[i+1]*...*M[j] for $1 \le i \le j \le N$.
m(i,j) = 0, if i = j;
= MIN (m(i,k)+m(k+1,j)+r(i-1)*r(k)*r(j) if $j > i$ and $i \le k < j$,
where r[i-1] and r[i] correspond to the dimension of M[i]. This method calculates m(i,j)'s in order of increasing difference in the subscripts. We begin by calculating m(i,i) for all 'i', then m(i,i+1) for all 'i' and so on. In this way, the terms m(i,k) and m(k+1,j) in calculating m(i,j) are made available. An order in which the multiplication may be done can be determined by recording, for each table entry of m(i,j), a value of 'k' which gives rise to the minimum m(i,j). Given the vector containing the structures corresponding to each matrix, of each matrix the pointer to each matrix, the dimension of each matrix and the number of matrices, this function evaluates the product of matrices using the order of evaluation of the product determined by dynamic programming approach. It returns a pointer to the structure which contains the pointer and the dimension of the resultant matrix.

Examples

1) Let M = M1 * M2 * M3, where dimensions of M1 = [10x20], M2 = [20x50], M3 = [50x15]. We then form a array of structures of type struct xx {float **x; int r,c,flag;}, where 'x' corresponds to the pointer to the matrix and 'r', 'c' fields corresponds to the dimension of 'x'. The field 'flag' must be initialised to '1' before invoking this routine. For the above case, three structures store the information about M1,M2 and M3 which are loaded in to a vector of dimension equal to the number of matrices to be multiplied.
mmult(data,3);
This program line evaluates the product of 3 matrices, whose structures are given in vector 'data', and returns a pointer to a similar structure as above, which contains the pointer to and the dimension of the resultant matrix 'M'.
Test program: *xmmult.c*

Related modules

None

Module name

orddit.c (R) - Routine to obtain half-tone images from gray-level images using ordered dither.

void	orddit(int **image, int nrow, int ncol, int lngl);

Inputs

image	two-dimensional array of integers, each integer representing the gray level corresponding to the row and column indices
nrow	number of rows in the image
ncol	number of columns in the image
lngl	logarithm to the base 2 of the number of gray levels in the image

Outputs

image	binary-valued two-dimensional matrix representing the desired half-tone image

Processing

This routine does an in-place screening operation on the input gray-level image to produce its half-tone counterpart. The half-toning is done by a process called ordered dispersed-dot dithering. In this process each k x k block of the image is compared against a k x k threshold array containing the numbers 0,1,..,number of gray levels-1. If the comparision shows that the image value is less than the threshold value, then the image value is set to 0; otherwise the image value is set to 1.

Examples

1) orddit (image,16,640,7);
This example obtains the half-tone counterpart of a 128 gray-level 16 x 640 image. The result is stored in-place.
Test program: *xorddit.c*

Related modules

None

Module name

ran1.c (S) - Routine to generate a uniformly distributed random variable.

float	ran1(int *idum);

Inputs

idum	initializing variable to ran1 routine. If this variable is negative, the routine re-initializes itself. Upon initialization, the routine returns a value of of idum = +1. Thus, on subsequent calls, the user need not specifically set this input.

Outputs

ran1	floating-point number uniformly distributed between 0 and 1.

Processing

This routine is used to generate uniformly distributed random numbers. If the argument 'idum' is negative, the random number generator is reinitialized; otherwise, a new random number is generated. The routine combines three linear congruential generators to produce the output. The first two generators are used to provide the upper and lower bits of a number. The third generator is used to index a 97 word-long array which yields the random number to be output. The combined outputs of the first two generators are inserted into the array to replace the number extracted.

Examples

```
1) idum=-1;
   ran1(&idum);
```
This example initializes the random number generator and returns the first random number.

```
2) ran1(&idum);
```
This example is used on subsequent calls to ran1.
Test program: *xran1.c*

Related modules

expdev.c, gasdev.c

Module name

rbit.c (S) - Routine to generate a random bit sequence using a maximum length shift register.

int rbit(long int *shfg);

Inputs

shfg	Initial value of shift register used by ran1 routine. On subsequent calls, the user need not specifically set this input.

Outputs:

rbit	Integer that is either 0 or 1.

Processing

This routine is used to generate a pseudo-random sequence of bits using a maximum length shift register.

Examples

```
1) shfg=1;
   rbit(&shfg);
```
This example initializes the random bit generator's shift register to 1 and returns the first random bit.

```
2) rbit(&shfg);
```
This example is used on subsequent calls to rbit.
Test program: *xrbit.c*

Related modules

none

Module name

rfft.c (R) - Routine to compute the first N/2 + 1 complex components of the Discrete Fourier Transform (DFT) of a real-valued data sequence of length N.

```
void rfft( float *twiddle, float *data, int N);
```

Inputs

twiddle	array of length 5N/4 of single-precision floating-point numbers containing the quantities $\cos(2\pi k/N)$, k = 0,1,...,(5N/4)-1.
data	array of single-precision floating-point numbers constituting the real-valued input data sequence.
N	length of the number of points in the input sequence, where N is a power of 2.

Outputs

data	array of length N (2 real and (N-1)/2 complex single-precision floating-point numbers) constituting the first (N/2)+1 components of the DFT of the real-valued input data. The symmetry property X[k] = conjugate of X[N-k] may be used to obtain the remaining (N/2)-1 Fourier components. The components are stored in the array 'data' in the following order:
	data[0]=Re{X[0]} (note that Im{X[0]}=0),
	data[1]=Re{X[N/2]} (note that Im{X[N/2]=0),
	data[2m]=Re{X[m]}, m=1,2,...,(N/2)-1,
	data[2m+1]=Im{X[m]},m=1,2,...,(N/2)-1.

Processing

This routine computes the DFT of a real input sequence, x[m], m=0,1,...,N-1. The first part of the routine considers the input to be a complex sequence,
y[n]=x[2n]+jx[2n+1], n=0,1,...,(N/2)-1, and computes an in-place DFT, Y[k], k=0,1,...,(N/2)-1.
The second part of the routine computes X(k), k=0,1,...,N/2, from Y[k], k=0,1,...,(N/2)-1.
Since the dc (0th) and the N/2th frequency components of the DFT of the real valued sequence are real, the imaginary part of X[0] is loaded with X[N/2] which makes the computation of the inverse possible.

Examples

```
1) gentwd (twiddle,32);
   rfft (twiddle,data,32);
```

This computes the DFT of 32-point real-valued sequence, and although the data array contains only 17 Fourier components, 2 of them real and 15 of them complex, the remaining 15 complex components can be computed using:
X[32-k] = conj{X[k]}, k=1,2,...,15,
i.e., Re{X[32-k]} = data[2k], k=1,2,...,15, and
Im{X[32-k]} = -data[2k+1], k=1,2,...,15.
Test program: *xrfft.c*

Related modules

gentwd.c, fft.c, irfft.c, twofft.c, itwofft.c

Module name

sinft.c (R) - Routine to compute the Discrete Sine Transform (DST) of a real-valued data record.

void sinft(float *twiddle, float *data, int N);

Inputs

twiddle	array of length 5N/2 of single-precision floating-point numbers containing the quantities $\cos(2\pi k/(2N))$, $k=0,1,\ldots,(5N/2)-1$.
data	array of single-precision floating-point numbers constituting the real-valued input data sequence.
N	length of the number of points in the input sequence, where N is a power of 2. Note that for the sine transform the first point is not included, and is taken to be zero.

Outputs:

data	array of length N, the first point always being zero, and the remaining N-1 points constituting the components of the sine transform of the real-valued input data.

Processing

This routine computes the sine transform of a real input sequence, $x[n]$. The first part of the routine forms the sequence,
$y[n]=(x[n]+x[N-n])(\sin(n\pi/N)+(x[n]-x[N-n])/2$, $n=1,2,\ldots,N-1$,
$y[0]=0$,
and computes the DFT, $Y[k]$, $k=0,1,\ldots,(N/2)-1$, using a routine identical to 'rfft' except for the indexing into the twiddle factor array. The second part of the routine computes the sine transform of $x[n]$, $S(m)$, $m=1,\ldots,N-1$, from $Y[k]$, $k=0,1,\ldots,(N/2)-1$ as follows:
$S[2k+1]=S[2k-1]+\mathrm{Re}\{Y[k]\}$, $k=1,2,\ldots,(N/2)-1$,
where the starting point of the recursion is
$S[1]=\mathrm{Re}\{Y[0]\}/2$,
and $S[2k]=-\mathrm{Im}\{Y[k]\}$, $k=1,2,\ldots,(N/2)-1$,
and where by convention $S[0]=0$.

Examples

1) gentwd (twiddle,64);
   sinft (twiddle,data,32);

This computes the sine transform of a 31-point real-valued sequence, stored in locations indexed from 1 to 31, with the location corresponding to the zeroth index being set to 0. The output has the location corresponding to the zeroth index also set to zero, while the remaining 31 locations of the array 'data' contain the sine transform of the data.

2) sinft (twiddle,data,32);

This computes the inverse sine transform of the sequence that is output in example 1, except for a scale factor. If the exact inverse sine transform is desired, then the result sequence should be multiplied by 1/16 (=2/32).

Test program: *xsinft.c*

Related modules

gentwd.c, fft.c, rfft.c, cosft.c

Module name

synd.c (R) - Routine to compute the syndrome from the received vector.

void	synd (int *rcv, int *zech, int *synv, int blen, int cerc);

Inputs

rcv	vector containing the received polynomial, left-justified and packed 8*sizeof(int) bits to the word.
zech	array of integers indexed by $i=0,1,...,2^{(m-1)}$, containing the Zech's logarithm values defined by $\alpha^{zech[i]} = XOR(1,\alpha^i)$.
blen	number of bits in the block.
cerc	number of errors correctable by the code.

Outputs

synv	vector of integers, 2*cerc long, containing the syndrome of the received codeword.

Processing

The odd syndromes of the received codeword by substituting $x=\alpha^{2k-1}$, $k=1,2,...,t$, into rcv(x). The even syndromes, S[2k] may be calculated using $S[2k]=S[k]^2$, $k=1,2,...,t$ for binary codes.

Examples

1) synd(rcv,zech,synv,127,4);

This example calculates the syndrome of rcv[x] in the vector 'synv' (length=8) for the 4-error correcting (127,99) BCH code.
Test program: *xbch.c*

Related modules

ffmul.c, ffadd.c, elcf.c

Module name

twoddct.c (R) - Routine to compute the two-dimension Discrete Cosine Transform (2-D DCT) of a real-valued two-dimensional array.

void twoddct (float *twiddle, float **M, int N);

Inputs

twiddle	array of length 5N/2 of single- precision floating-point numbers containing the quantities cos(πk)/N, k=0,1,...,(5N/2)-1.
M	N x N matrix of single-precision floating- point numbers constituting the real input data.
N	length of each row (equal to the number of columns), where N is a power 2.

Outputs

M	N x N matrix of single-precision floating- point numbers constituting the transpose of the two-dimensional DCT.

Processing

This routine does an in-place 2-D DCT of a real input matrix, i.e., the result for each stage is stored in the same array as the previous stage.
The first part of the routine takes the N row- wise DCT's of the N x N matrix by calling the routine 'cosft' N times. After each transform, each row element is multipled by cos((πj)/(2N)), where j is the column index.
The next part of routine transposes the N x N matrix so that the N column-wise DFT's may be taken using 'cosft'.

The final part of the routine takes the N column-wise DCT's. After each DCT, the row elements are multiplied in the same fashion as in step 1. Note that the result is the transpose of the desired DCT. The user will have to transpose the result if he wants the DCT in the proper order, but this will not be necessary for most applications of this routine.

Examples

1) gentwd (twiddle,64);
   twoddct (twiddle,M,32);

This computes the transpose of the DCT of a 32 x 32 point matrix, M, and stores the result in M. 'twiddle' is the array of twiddle factors which are calculated using routine 'gentwd'.
Test program: *(see source code)*

Related modules

gentwd.c, cosft.c

Module name

twodfft.c (R) - Routine to compute the two-dimensional Discrete Fourier Transform (2-D FFT) of a complex-valued two-dimensional array.

void	twodfft(float *twiddle, float **M1, float **M2, int N);

Inputs

twiddle	array of length 5N/4 of single-precision floating-point numbers containing the quantities cos(2πk)/N, k=0,1,...,(5N/4)-1.
M1	N x N matrix of single-precision floating-point numbers constituting the real part of input data.
M2	N x N matrix of single-precision floating-point numbers constituting the imaginary part of input data.
N	length of each row (equal to the number of columns), where N is a power 2.

Outputs

M1	N x N matrix of single-precision floating-point numbers constituting the transpose of the real part of the DFT.
M2	N x N matrix of single-precision floating-point numbers constituting the transpose of the imaginary part of the DFT.

Processing

This routine does an in-place 2-D DFT of a complex input matrix, i.e., the result for each stage is stored in the same array as the previous stage.
The first part of the routine takes the N row- wise DFT's of the N x N matrix by calling the routine 'fft' N times.
The next part of routine transposes the N x N matrix so that the N column-wise DFT's may be taken using 'fft'.The final part of the routine takes the N column- wise DFT's. Note that the result is the transpose of the desired DFT. You will have to transpose the result if you want the DFT in the proper order, but this will not be necessary for most applications of this routine.

Examples

1) gentwd (twiddle,32);
   twodfft (twiddle,MR,MI,32);

This computes the transpose of the DFT of a 32 x 32 point complex matrix contained in MR and MI and stores the result in MR and MI. 'twiddle' is the array of twiddle factors which are calculated using routine 'gentwd'.

2) twodfft (twiddle,MI,MR,N);

The inverse DFT of a complex sequence can be computed (to within a scale factor) by calling the fft routine with the real and imaginary data reversed in order. Note that MI and MR should be the matrix of transposed spatial frequency components, i.e., the pair of program lines,

twodfft (twiddle,MR,MI,N);
twodfft (twiddle,MI,MR,N);

produce the original input sequence scaled by a factor of N^2. Also note that if the exact inverse is desired, then each of the components of datai and datar should be multiplied by $1/N^2$.

Test program: *(see source code)*

Related modules

gentwd.c, fft.c

Module name

twofft.c (R) - Routine to calculate the first N/2 + 1 components of the Discrete Fourier Transforms (DFT's) of two real-valued data sequences of length N simultaneously.

```
void twofft( float *twiddle, float *data1, float *data2, int N);
```

Inputs

twiddle	array of length 5N/4 of single-precision floating-point numbers containing the quantities cos(2πk/N), k = 0,1,...,(5N/4)-1.
data1	array of single-precision floating-point numbers constituting the first real-valued input data sequence.
data2	array of single-precision floating-point numbers constituting the second real-valued input data sequence.
N	length of the number of points in the input sequences, where N is a power of 2.

Outputs

data1	array of length N (2 real and (N-1)/2 complex single-precision floating-point numbers) constituting the first (N/2)+1 components of the DFT of the first real-valued input data. The symmetry property X[k] = conjugate of X[N/2-k] may be used to obtain the remaining (N/2)-1 Fourier components. The components are stored in the array 'data1' in the following order:
	data1[0]=Re{X[0]} (note that Im{X[0]}=0),
	data1[N/2]=Re{X[N/2]} (note that Im{X[N/2]=0),
	data1[m]=Re{X[m]}, m=1,2,...,(N/2)-1,
	data1[N-m]=Im{X[m]}, m=1,2,...,(N/2)-1.

data2	array of length N (2 real and (N-1)/2 complex single-precision floating-point numbers) constituting the first (N/2)+1 components of the DFT of the second real-valued input data. The symmetry property Y[k] = conjugate of Y[N/2-k] may be used to obtain the remaining (N/2)-1 Fourier components. The components are stored in the array 'data2' in the following order:
	data2[0]=Re{Y[0]} (note that Im{Y[0]}=0),
	data2[N/2]=Re{Y[N/2]} (note that Im{Y[N/2]=0),
	data2[m]=Re{Y[m]}, m=1,2,...,(N/2)-1,
	data2[N-m]=Im{Y[m]}, m=1,2,...,(N/2)-1.

Processing

This routine computes the DFT of two real input sequences simultaneously. The first part of the routine considers the input to be a complex sequence, z[n]=x[n]+jy[n], n=0,1,...,N-1, and computes an in-place DFT, Z[k], k=0,1,...,N-1. The second part of the routine computes X[k] and Y[k], k=0,1,...,N/2, from Z[k], k=0,1,...,N-1. Since the dc (0th) and the N/2th frequency components of the DFT's of the real valued sequences are real, and the remaining (N/2)-1 components complex, N locations are adequate to store the DFT sequences of x[n] and y[n] in an invertible form.

Examples

1) gentwd (twiddle,32);
   twofft (twiddle,data1,data2,32);

This computes the DFT of two 32-point real-valued sequences simultaneously, and although the data array contains only 17 Fourier components, 2 of them real and 15 of them complex, the remaining 15 complex components can be computed using:
X[32-k] = conj{X[k]}, k=1,2,...,15,
i.e., Re{X[32-k]} = data1[k], k=1,2,...,15,
and Im{X[32-k]} = -data1[32-k], k=1,2,...,15,
and Y[32-k] = conj{Y[k]}, k=1,2,...,15,
i.e., Re{Y[32-k]} = data2[k], k=1,2,...,15,
and Im{Y[32-k]} = -data2[32-k], k=1,2,...,15.
Test program: *x2fft.c*

Related modules

gentwd.c, fft.c, rfft.c, irfft.c, itwofft.c

Module name

vitbi.c (R) - Routine to implement a Viterbi decoder

int vitbi(float *mf, unsigned int **perm, float *stamet, long *phist, float *bsm, int *skip, int nstates);

Inputs

mf	array of length=1/(rate of code) inverted and normalized metrics.
perm	integer matrix of dimension nstates x 2 containing the symbol corresponding to each 'butterfly' of the Viterbi decoder.
stamet	floating-point vector of length=nstates containing the state metrics of the Viterbi decoder.
phist	long integer vector containing the (8*sizeof(int)) previous bits (the path history) corresponding to each state.
bsm	best (minimum) state metric (should be initialized to MAXMET on entry to 'vitbi').
skip	defines the difference (modulo nstates) between the two points on a 'butterfly'.
nstates	defines the number of states ($2^{(\text{constraint length-1})}$) of the convolutional decoder.

Outputs

stamet	floating-point vector of length=nstates containing the updated state metrics of the Viterbi decoder.
phist	long integer vector containing the updated (8*sizeof(int)) previous bits (the path history) corresponding to each state.
bsm	best (minimum) state metric after the current pass through the Viterbi decoder. It is used to "normalize" the inverted metrics of the next pass.
skip	defines the difference (modulo nstates) between the two points on a 'butterfly' for the next pass.
vitbi	the decoded bit for the current pass (this bit corresponds to the symbol received 32+(constraint length-1) bauds before the current baud).

Processing

The convolutional decoder calls the add-compare-select routine for each of the (nstates/2) 'butterflies' in the viterbi decoder. It then updates the value 'skip' for the next pass's 'butterflies'. The best state metric (bsm) is output in order that the next pass's branch metrics may be 'normalized' (by subtracting bsm from them). Finally, it outputs the oldest bit in the path history corresponding to the state whose state metric is 'bsm'.

Examples

```
1) for (i=0; i<(1<<RATE); i++)   mf[i]=MAXMET-mf[i]-bsm;
   bsm=MAXMET;
   obit=vitbi(mf,perm,stamet,phist,&bsm,&skip,nstates);
```

This example updates the state metrics in 'stamet' using the branch metrics in 'bsm'. It also updates the path history table 'phist'. The smallest state metric is returned in 'bsm', while the outpqut bit is returned in 'vitbi'.
Test program: *xconv.c*

Related modules

cncod.c, acs.c

PRODUCT REGISTRATION

****

Please retain this copy for your records and return the next page to us with complete information

****

The Product you have Purchased Is:________________
Product serial number is:___________________________

Technical Support:

You will receive free product updates and free technical support for next 3 months. For technical support contact:

Sonitech International, Inc.
83 Fullerbrook Road
Wellesley, MA 02181, USA
Telephone: (617)235-6824, Fax: (617)235-2531

Replacement Policy:

We will replace your software diskettes free of charge within 45 days of purchase if they prove defective. You must call or write to us first indicating the problem and we will respond to you by telephone or writing within 3 days after hearing from you.

Registration:

For us to support and to provide you with timely updates of the software, please completely fill the form on the next page and mail to:

Sonitech International Inc.
83 Fullerbrook Road
Wellesley, MA 02181, USA

*** This page blank ***

REGISTRATION CARD

We have read the *Sonitech International Inc.'s* program license agreement and agree to abide by the terms and conditions contained therein.

______________________ ____________________________
Signature Name of Customer

Name____
Title____
Organization__________________________________
Address______________________________________
Address______________________________________
City____
State_________________Code__________________
Country______________________________________
Telephone or Telex_____________________________

Purchase Information:

Product Name:_________________________________
Product Serial Number: 1048

Software Purchased From:

__ *Sonitech International Inc.*
__ Distributor
Name:_____________________________________
Country:___________________________________
Address:___________________________________
__ Other

How did you first hear about our Products:
__ From Distributor
__ Mail
__ Advertising
__ Editorial
__ University

Please comment (on back of this page) about this package. Thank you.

Comments:____________________________________

____

____

WARRANTY FORM

This product is warranted against defects in materials and workmanship for 45 days. The Warranty is effective from date of purchase. Sonitech International Inc. (Sonitech) will repair or replace, at its option, any product found to be defective during the Warranty period.

EXCEPT TO THE EXTENT PROHIBITED BY APPLICABLE LAW, NO OTHER WARRANTIES, WHETHER EXPRESSED OR IMPLIED, INCLUDING WARRANTIES OF MERCHANTABILITY AN FITNESS FOR A PARTICULAR PURPOSE, SHALL APPLY TO THIS PRODUCT; UNDER NO CIRCUMSTANCES SHALL SONITECH BE LIABLE FOR CONSEQUENTIAL DAMAGES SUSTAINED IN CONNECTION WITH SAID PRODUCT AND NEITHER ASSUMES NOR AUTHORIZES ANY REPRESENTATIVE OR OTHER PERSON TO ASSUME FOR IT ANY OBLIGATION OR LIABILITY OTHER THAN SUCH AS IS EXPRESSLY SET FORTH HEREIN.

Before returning this product, you must receive a Return Material Authorization (RAM). Returned product will not be accepted without this authorization.

SOFTWARE LICENSE AGREEMENT

The program(s) delivered with this Agreement are sold only on the condition that the purchaser agrees to the terms and conditions of this Agreement. READ THIS AGREEMENT CAREFULLY. If you do not agree, return the packaged program UNOPENED to your distributor or dealer and your purchase price will be refunded. If you agree, fill out and sign the Registration Form and RETURN to us by mail.

Sonitech International Inc. (hereinafter called "Sonitech") agrees to grant and the Customer agrees to accept, subject to the following terms and conditions, a personal, nonexclusive and nontransferable license to use the propriety program(s) (hereinafter called the "Program") of Sonitech International Inc. delivered with this agreement.

The program includes executable software, object files, and application programs. The program does not include data files, sample executables, and demo programs.

License

The license granted hereunder authorizes the Customer to use the Program in machine readable form on any single computer system (hereinafter called the "System"). A separate license is required for each System on which the Program will be use.

Copy

The program may only be copied, in whole or in part, in printed or machine readable form, for the use by the Customer on the System, to understand the contents of the Program, for back-up purposes, or for archive purposes; provided, however, that no more than two (2) copies shall be in existence with respect to any Program at any one time without prior written consent of Sonitech International Inc.

Term

This Agreement shall become effective as of the date of shipment of the Program from Sonitech International to the Customer, and shall continue in force until terminated by either party hereto pursuant to terms below:

The customer may terminate this Agreement by returning the Program unopened within 15 days, the Customer of delivery.

Miscellaneous

The rights and benefits of the Customer hereunder shall not be assigned or transferred in any manner whatsoever.

The validity and construction of this Agreement shall be governed by the laws of the State of Massachusetts. The parties shall attempt to settle disputes, controversies or differences which may arise out of or in relation to or in connection with this Agreement.