# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGPRO OVERVIEW</td>
<td>4</td>
</tr>
<tr>
<td>SIGPRO FUNCTION DESCRIPTIONS</td>
<td>5</td>
</tr>
<tr>
<td>sp_bessel</td>
<td>5</td>
</tr>
<tr>
<td>sp_butter</td>
<td>6</td>
</tr>
<tr>
<td>sp_cdb</td>
<td>7</td>
</tr>
<tr>
<td>sp_chirp</td>
<td>8</td>
</tr>
<tr>
<td>sp_cgd</td>
<td>9</td>
</tr>
<tr>
<td>sp_cheby</td>
<td>10</td>
</tr>
<tr>
<td>sp_cmagsq</td>
<td>11</td>
</tr>
<tr>
<td>sp_convert</td>
<td>12</td>
</tr>
<tr>
<td>sp_copy</td>
<td>13</td>
</tr>
<tr>
<td>sp_eph</td>
<td>14</td>
</tr>
<tr>
<td>sp_crfft</td>
<td>15</td>
</tr>
<tr>
<td>sp_cvadd</td>
<td>16</td>
</tr>
<tr>
<td>sp_cvdiv</td>
<td>17</td>
</tr>
<tr>
<td>sp_cvmul</td>
<td>18</td>
</tr>
<tr>
<td>sp_cvsub</td>
<td>19</td>
</tr>
<tr>
<td>sp_fft</td>
<td>20</td>
</tr>
<tr>
<td>sp_fftfilt</td>
<td>21</td>
</tr>
<tr>
<td>sp_fftfiltz</td>
<td>22</td>
</tr>
<tr>
<td>sp_filter</td>
<td>23</td>
</tr>
<tr>
<td>sp_filterz</td>
<td>24</td>
</tr>
<tr>
<td>sp_firdb</td>
<td>25</td>
</tr>
<tr>
<td>sp_fmins</td>
<td>26</td>
</tr>
<tr>
<td>sp_freqshape</td>
<td>27</td>
</tr>
<tr>
<td>sp_freqz</td>
<td>28</td>
</tr>
<tr>
<td>sp_freqshp</td>
<td>29</td>
</tr>
<tr>
<td>sp_ifft</td>
<td>30</td>
</tr>
<tr>
<td>sp_interp</td>
<td>31</td>
</tr>
<tr>
<td>sp_linspace</td>
<td>32</td>
</tr>
<tr>
<td>sp_mat_append</td>
<td>33</td>
</tr>
<tr>
<td>sp_mat_fetch</td>
<td>34</td>
</tr>
<tr>
<td>sp_mat_load</td>
<td>35</td>
</tr>
<tr>
<td>sp_mat_save</td>
<td>36</td>
</tr>
<tr>
<td>sp_mat_size</td>
<td>37</td>
</tr>
<tr>
<td>sp_mat_version</td>
<td>38</td>
</tr>
<tr>
<td>sp_mat_whos</td>
<td>39</td>
</tr>
<tr>
<td>sp_nxtpow2</td>
<td>40</td>
</tr>
<tr>
<td>sp_rand</td>
<td>41</td>
</tr>
<tr>
<td>sp_randflat</td>
<td>42</td>
</tr>
<tr>
<td>sp_randn</td>
<td>43</td>
</tr>
<tr>
<td>sp_randseed</td>
<td>44</td>
</tr>
<tr>
<td>sp_rcfft</td>
<td>45</td>
</tr>
<tr>
<td>sp_sadd</td>
<td>46</td>
</tr>
</tbody>
</table>
Signal Processing Library

sp_sma........................................................................................................................... 47
sp_smul ......................................................................................................................... 48
sp_transfer ..................................................................................................................... 49
sp_unwrap ..................................................................................................................... 50
sp_vadd ......................................................................................................................... 51
sp_var_alloc .................................................................................................................. 52
sp_var_clear .................................................................................................................. 53
sp_var_clear_all .......................................................................................................... 54
sp_var_copy .................................................................................................................. 55
sp_var_find ................................................................................................................... 56
sp_var_float ................................................................................................................... 57
sp_var_f4 ....................................................................................................................... 58
sp_var_f8 ....................................................................................................................... 59
sp_var_i2 ....................................................................................................................... 60
sp_var_i4 ....................................................................................................................... 61
sp_var_set ..................................................................................................................... 62
sp_var_size .................................................................................................................... 63
sp_version ..................................................................................................................... 64
sp_vdot .......................................................................................................................... 65
sp_vdiv .......................................................................................................................... 66
sp_vmax ......................................................................................................................... 68
sp_vmin ......................................................................................................................... 69
sp_vmul ......................................................................................................................... 70
sp_vsub ........................................................................................................................ 71
sp_wav_info .................................................................................................................. 73
sp_wav_read .................................................................................................................. 74
sp_window ..................................................................................................................... 75
sp_zero .......................................................................................................................... 76
Appendix A. MAT and VAR functions ............................................................................ 77
Appendix B. OPT structure ............................................................................................... 78
Appendix C. Test Programs .............................................................................................. 79
SIGPRO LIBRARY OVERVIEW

This reference manual describes a signal processing library designed to assist in the development of auditory research software. Current functions include random number generators, fft, inverse fft, frequency shaping (filtering), and sample rate conversion. Limited support has been added for loading and saving binary (MAT) files. The current version of the SIGPRO library is 0.22. The most version of the SIGPRO source code and documentation can be downloaded from http://audres.org/rc/sigpro/.
SIGPRO FUNCTION DESCRIPTIONS

sp_bessel
Bessel-style IIR filter design.

(void) sp_bessel(float *b, float *a, int n, float *wn, int ft)

Parameters

- **b**: input (numerator) coefficients
- **a**: output (denominator) coefficients
- **n**: order of filter
- **wn**: cutoff frequency re Nyquist frequency
- **ft**: filter type

Return Value

none

Remarks

Filter design is based on a bilinear transformation of the classic analog Bessel (type I) filter. The filter type specifies whether the filter is low-pass (ft=0), high-pass (ft=1), band-pass (ft=2), or band-stop (ft=3). The cutoff frequency is divided by half the sampling rate (i.e., the Nyquist frequency). The number of elements in the input and output coefficient arrays will be the order of the filter plus 1 for low-pass or high-pass filters (ft=0 or ft=1). The number of elements in the input and output coefficient arrays will be twice the order of the filter plus 1 for band-pass or band-stop filters (ft=0 or ft=1). Only one cutoff frequency is needed when the filter type is low-pass or high-pass. Two cutoff frequencies are needed when the filter type is band-pass or band-stop. The input and output coefficient arrays may be used to perform filtering with sp_rcfft.

See Also

sp_butter, sp_cheby, sp_filter
**sp_butter**
Butterworth-style IIR filter design.

(void) **sp_butter**(float *b, float *a, int n, float *wn, int ft)

**Parameters**
- b: input (numerator) coefficients
- a: output (denominator) coefficients
- n: order of filter
- wn: cutoff frequency re Nyquist frequency
- ft: filter type

**Return Value**
none

**Remarks**
Filter design is based on a bilinear transformation of the classic analog Chebyshev (type I) filter. The filter type specifies whether the filter is low-pass (ft=0), high-pass (ft=1), band-pass (ft=2), or band-stop (ft=3). The cutoff frequency is divided by half the sampling rate (i.e., the Nyquist frequency). The number of elements in the input and output coefficient arrays will be the order of the filter plus 1 for low-pass or high-pass filters (ft=0 or ft=1). The number of elements in the input and output coefficient arrays will be twice the order of the filter plus 1 for band-pass or band-stop filters (ft=0 or ft=1). Only one cutoff frequency is needed when the filter type is low-pass or high-pass. Two cutoff frequencies are needed when the filter type is band-pass or band-stop. The input and output coefficient arrays may be used to perform filtering with **sp_rcfft**.

**See Also**
- **sp_bessel**, **sp_cheby**, **sp_filter**
**sp_cdb**

Returns real decibels for a complex input.

```
(void) sp_cdb(float *x, float *db, int n)
```

**Parameters**

- `x`  complex\(^1\) input array
- `db` real output array (dB)
- `n` output array size

**Return Value**

none

**Remarks**

Useful for obtaining spectral magnitude from the complex spectral values returned by **sp_rcfft**. Output array has the same number of elements as the input array, but is half the size because it has no imaginary components.

\(^1\) The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
sp_chirp
Generate frequency-sweep waveform.

(void) sp_chirp(float *x, int n)

Parameters
    x        output array
    n        array size

Return Value
    none

Remarks
    The waveform generated in the x array is a sine wave with instantaneous
    frequency that increases linearly with time from the lowest to the highest possible
    frequency. The maximum amplitude of this frequency-sweep tone is one.


**sp_cgd**
Returns real group delay for a complex input.

**(void) sp_cgd(float *, float *, int, double)**

**Parameters**
- **x**: complex input array
- **gd**: real output array (s)
- **n**: output array size
- **df**: frequency increment (Hz)

**Return Value**
none

**Remarks**
Useful for obtaining group delay from the complex spectral values returned by **sp_rcfft**. Output array has the same number of elements as the input array, but is half the size because it has no imaginary components.

---

2 The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
**sp_cheby**

Chebyshev-style IIR filter design.

( void ) **sp_cheby**(float *b, float *a, int n, float *wn, int ft, double rip)

**Parameters**

- **b** : input (numerator) coefficients
- **a** : output (denominator) coefficients
- **n** : order of filter
- **wn** : cutoff frequency \( re \) Nyquist frequency
- **ft** : filter type
- **rip** : pass-band ripple (dB)

**Return Value**

none

**Remarks**

Filter design is based on a bilinear transformation of the classic analog Chebyshev (type I) filter. The filter type specifies whether the filter is low-pass \((ft=0)\), high-pass \((ft=1)\), band-pass \((ft=2)\), or band-stop \((ft=3)\). The cutoff frequency is divided by half the sampling rate \(i.e., the Nyquist frequency\). The number of elements in the input and output coefficient arrays will be the order of the filter plus 1 for low-pass or high-pass filters \((ft=0\ or\ ft=1)\). The number of elements in the input and output coefficient arrays will be twice the order of the filter plus 1 for band-pass or band-stop filters \((ft=0\ or\ ft=1)\). Only one cutoff frequency is needed when the filter type is low-pass or high-pass. Two cutoff frequencies are needed when the filter type is band-pass or band-stop. The input and output coefficient arrays may be used to perform filtering with **sp_rcefft**.

**See Also**

- **sp_bessel**, **sp_butter**, **sp_filter**
**sp_cmagsq**
Complex magnitude squared.

( void ) sp_cmagsq( float *x, float *y, int n )

**Parameters**
- **x** complex input array
- **y** complex output array
- **n** complex array size

**Return Value**
None

**Remarks**
The input and output arrays float variables with alternating real and imaginary parts of complex values. The arrays size is the number of real and imaginary pairs in each array. On exit, the real part of the output arrays contains the sum of the squares of the real and imaginary parts of the corresponding complex value in the input array. The imaginary parts of the output array are all set to zero.
**sp_convert**

Convert sampling rate.

(int) **sp_convert** (float *x1, int n1, float *x2, int n2, double rr, int wrap)

**Parameters**

- **x1**: input waveform
- **n1**: input size
- **x2**: output waveform
- **n2**: output size
- **rr**: sample rate ratio
- **wrap**: wrap flag (0=no, 1=yes)

**Return Value**

- 0: Success
- 1: Null values passed

**Remarks**

Uses the waveform in **x1** and sinc-function interpolation to create a waveform in **x2** with **n2** number of samples. Set parameter **rr** to the desired ratio of output sampling rate to input sampling rate or set **rr=0** to select a sampling rate ratio equal to **n2/n1**. The waveform in **x1** is assumed to be zero outside the specified range when **wrap=0**, or assumed to be periodic when **wrap=1**.

April 17, 2011  12
Signal Processing Library

*sp_copy*
Copies an array

(void) *sp_copy*(float *x, float *y, int n)

**Parameters**
- x: input array
- y: output array
- n: array size

**Return Value**
none

**Remarks**
Copies x into y.
$sp_{cph}$

Returns real phase for a complex input.

(void) $sp_{cph}$(float *x, float *ph, int n)

**Parameters**

- x: complex³ input array
- ph: real output array (cycles)
- n: complex array size

**Return Value**

none

**Remarks**

Useful for obtaining spectral phase from the complex spectral values returned by $sp_{rcfft}$. Use $sp_{unwrap}$ to unwrap phase. Output array has the same number of elements as the input array, but is half the size because it has no imaginary components.

³ The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
**sp_crfif**
Complex to real inverse FFT

(int) **sp_crfif**(float *x, int n)

**Parameters**

- **x**: complex⁴ input, real output array
- **n**: output size + 2

**Return Value**

- 0: success

**Remarks**
Performs inverse (complex to real) FFT on **x** (in place). If **n** is a power of two, then a fast Fourier transform is used; otherwise the Fourier transform is slow. The input **x** is expected to contain \( n/2+1 \) complex values (i.e., \( x[0]=\text{real}, \ x[1]=\text{imaginary}, \) etc.). Returned in **x** are **n** real values. The **x** array size is **n+2**.

---

⁴ The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
Signal Processing Library

*sp_cvadd*
Complex-vector add

(void) *sp_cvadd*(float *x, float *y, float *z, int n)

**Parameters**
- **x** input array
- **y** input array
- **z** output array
- **n** array size

**Return Value**
none

**Remarks**
- Adds two complex vectors: \( z = x + y \).

The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
Signal Processing Library

\textit{sp_cvdiv}

Complex-vector divide

(int) \textit{sp_cvdiv}(float *x, float *y, float *z, int n)

\textbf{Parameters}

\begin{itemize}
    \item \textbf{x} \hspace{1cm} 	ext{input array}
    \item \textbf{y} \hspace{1cm} 	ext{input array}
    \item \textbf{z} \hspace{1cm} 	ext{output array}
    \item \textbf{n} \hspace{1cm} 	ext{array size}
\end{itemize}

\textbf{Return Value}

Number of divisions by zero (not performed). Values are returned for all non-zero divisions.

\textbf{Remarks}

Divides two complex vectors: \( z = x / y \).

The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
sp_cvmul
Complex-vector multiply

(void) sp_cvmul(float *x, float *y, float *z, int n)

Parameters
    x    input array
    y    input array
    z    output array
    n    array size

Return Value
none

Remarks
    Multiplies two complex vectors: z = x*y.

The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
**sp_cvsub**
Complex-vector subtract

( void ) **sp_cvsub** ( float * x, float * y, float * z, int n )

**Parameters**
- x : input array
- y : input array
- z : output array
- n : array size

**Return Value**
none

**Remarks**
Subtracts two complex vectors: \( z = x - y \).

The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
sp_fft
Complex to complex fft.

(int) sp_fft(float *x, int n)

Parameters

\[ x \] complex⁵ input, complex output array
\[ n \] input/output size

Return Value

0 success

Remarks

Performs FFT (in place) on \( x \). The input and output arrays are complex, with alternating real and imaginary values. If \( n \) is a power of two, then a fast Fourier transform is used; otherwise the Fourier transform is slow. The \( x \) array size is \( 2n \).

⁵ The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
sp_fftfilt
FIR filter using FFT-based overlap-add method.

(int) sp_fftfilt(float *b, int nb, float *x, int n, float *y, int wrap)

Parameters
  b    input-coefficient array
  nb   input-coefficient array size
  x    input data array
  n    input/output data array size
  y    output data array
  wrap wrap flag (0=no, 1=yes)

Return Value
  0    success

Remarks
  Performs FIR filter in place on x using FFTs. The input and output are assumed to be periodic when wrap=1.
**sp_fftfiltz**  
FIR filter using FFT-based overlap-add method with history.

(int) sp_fftfiltz(float *b, int nb, float *x, int n, float *y, float *z)

**Parameters**

- **b**: input-coefficient array  
- **nb**: input-coefficient array size  
- **x**: input data array  
- **n**: input/output data array size  
- **y**: output data array  
- **z**: history data array (size=nb)

**Return Value**

- 0: success

**Remarks**

Performs FIR filter in place on x using FFTs. The history data array contains output data that extends beyond the output data array.
Signal Processing Library

*sp_filter*
Filter data with recursive (IIR) or non-recursive (FIR) filter.

(int) `sp_filter(float *b, int nb, float *a, int na, float *x, float *y, int n)`

Parameters
- `b` input-coefficient array
- `nb` input-coefficient array size
- `a` output-coefficient array
- `na` output-coefficient array size
- `x` input data array
- `y` output data array
- `n` input/output array size

Return Value
- Error code

Remarks
A non-recursive (FIR) filter is specified by setting `a=NULL` and/or `na=0`. Recursive filter coefficients are normalized when `a[0]` is not equal to `1`. 
**sp_filterz**
Filter data with recursive (IIR) or non-recursive (FIR) filter with history.

(int) `sp_filterz(float *b, int nb, float *a, int na, float *x, float *y, int n, float *z)`

**Parameters**
- **b**: input-coefficient array
- **nb**: input-coefficient array size
- **a**: output-coefficient array
- **na**: output-coefficient array size
- **x**: input data array
- **y**: output data array
- **n**: input/output data array size
- **z**: history data array (size=nb)

**Return Value**
- Error code

**Remarks**
A non-recursive (FIR) filter is specified by setting `a=NULL` and/or `na=0`. Recursive filter coefficients are normalized when `a[0]` is not equal to 1. The history data array contains output data that extends into the input data (on input) or beyond the output data (on output).
**sp_firdb**
FIR frequency shape filter.

(int) `sp_firdb(float *b, int nb, float fs, float *ft, float *at, int nt)`

**Parameters**
- `b` : FIR waveform
- `nb` : FIR size
- `fs` : sampling frequency (Hz)
- `ft` : frequency table (Hz)
- `at` : attenuation table
- `nt` : table size

**Return Value**
- 0 : Success
- 1 : Table size too small
- 2 : `ft` order non-monotonic
- 3 : `ft` range not within 0 and `fs/2`

**Remarks**
Returns an impulse response of length n for an FIR filter with the specified frequency response. If `nb` is a power of two, then a fast Fourier transform is used; otherwise the Fourier transform is slow. Array `ft` contains the specific frequencies to shape. Array `at` contains dB attenuation values. The size of both `ft` and `at` arrays is `nt`. Array `ft` must have 0 as its first entry and `fs/2` as its last entry.
Signal Processing Library

**sp_fmins**
Search parameter space to find minimum value of function.

(int) sp_fmins(float *p, int n, double (*v)(float *), OPT *o)

**Parameters**
- p, Parameter array
- n, Array size
- (double *)(v*)(float *), Error function
- OPT *o Options

**Return Value**
- 0 Success
- 1 Too many parameters

**Remarks**
Uses the *simplex* method to find a set of parameters that minimizes the return value of a specified function. The parameter array must contain initial values on entry, which will be replaced by final values on return. The *error function* must accept a trial set of parameter values and return an error value, such as the sum of squared deviations. The option structure allows some control over iteration details or can be set to NULL. The OPT structure is described in Appendix B.
**sp_freqshape**
Performs frequency shaping on periodic input waveform.

(int) **sp_freqshape**(float *f, float *x, float *y, int n, float *ft, float *at, int nt)

**Parameters**
- f: FFT frequencies (Hz)
- x: input waveform
- y: output waveform
- n: waveform size
- ft: frequency table (Hz)
- at: attenuation table (dB)
- nt: table size

**Return Value**
- 0: Success
- 1: Table size too small
- 2: ft order non-monotonic
- 3: f outside range of ft

**Remarks**
Performs frequency shaping on x and returns the modified waveform in y. The size of both x and y arrays is n. If that size is a power of two, then the filtering, which uses an FFT, will be much faster. The input and output arrays are assumed to be periodic. Array f contains the frequencies of the spectrum of x and its size is n/2+1. Array ft contains the specific frequencies to shape. Array at contains dB attenuation values. The size of both ft and at arrays is nt. The range of ft must span f. The sp_freqshape function has been deprecated and replaced by the sp_frqshp function.

**See Also**
- sp_frqshp
**sp_freqz**

IIR filter transfer function.

(void) **sp_freqz**(float *b, int nb, float *a, int na, float *f, float *H, int nf, double fs)

**Parameters**

- **b**: input (numerator) coefficients
- **nb**: number of input coefficients
- **a**: output (denominator) coefficients
- **na**: number of output coefficients
- **f**: array of frequencies (Hz)
- **H**: complex⁶ transfer function \(H=b(z)/a(z)\)
- **nf**: number of frequencies
- **fs**: sampling rate (Hz)

**Return Value**

none

**Remarks**

A transfer function for the IIR filter defined by \(b\) and \(a\) is returned in \(H\) at each frequency listed in \(f\). The frequencies in \(f\) are in the same units as the sampling rate \(fs\).

**See Also**

- **sp_transfer**

---

⁶ The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
**sp_frqshp**
Performs frequency shaping on arbitrary input waveform.

(int) **sp_frqshp**(float *x, float *y, int n, double fs, float *fr, float *at, int nt, int wrap)

**Parameters**
- x: input waveform
- y: output waveform
- n: waveform size
- nf: FIR filter size
- fs: sampling frequency (Hz)
- fr: frequency table (Hz)
- at: attenuation table
- nt: table size
- wrap: wrap flag (0=no, 1=yes)

**Return Value**
- 0: Success
- 1: Table size too small
- 2: ft order non-monotonic
- 3: ft range outside 0 and fs/2

**Remarks**
Perform frequency shaping on x and returns the modified waveform in y. The size of both x and y arrays is n. Array ft contains the specific frequencies to shape. Array at contains dB attenuation values. The size of both ft and at arrays is nt. Array ft must have 0 as its first entry and fs/2 as its last entry. The waveform and FIR sizes (n and nf) are not required to be a power of two. The FFT size will be the power of 2 that is greater or equal to nf. The sp_frqshp function replaces the sp_freqshape function.
**sp_ifft**
Complex to complex inverse fft.

(int) **sp_ifft** (float *x, int n)

**Parameters**
- x: complex\(^7\) input, complex output array
- n: input/output size

**Return Value**
Error code

**Remarks**
Performs an inverse FFT on \(x\) (in place). The input and output arrays are complex, with alternating real and imaginary values. If \(n\) is a power of two, then a fast Fourier transform is used; otherwise the Fourier transform is slow. The \(x\) array size is \(2n\).

\(^7\)The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
**sp_interp**
Interpolates table values.

```
(int) sp_interp(float *x1, float *y1, int n1, float *x2, float *y2, int n2);
```

**Parameters**
- **x1**: table x
- **y1**: table y
- **n1**: table size
- **x2**: interpolate x
- **y2**: interpolate y
- **n2**: interpolate size

**Return Value**
- 0: Success
- 1: Table size too small
- 2: Table nonmonotonic

**Remarks**
Performs linear interpolation. The values in x1 and y1 are used to create a new set of values in x2 and y2 with n2 number of points.
Signal Processing Library

**sp_linspace**
Generates linearly spaced values

(void) \textbf{sp_linspace} (float \*x, int n, double a, double b)

**Parameters**
- \textbf{x} output array
- \textbf{n} array size
- \textbf{a} first value
- \textbf{b} last value

**Return Value**
none

**Remarks**
Returns \( n \) values linearly spaced between \( a \) and \( b \) in \( x \).
Signal Processing Library

**sp_mat_append**  
Appends variables to an existing MAT-format file.

(int) **sp_mat_append**(char *fn, VAR *vl)

**Parameters**  
- fn  file name  
- vl  variable list  

**Return Value**  
error code, which is zero for no errors  

**Remarks**  
Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**  
  
**sp_mat_whos, sp_mat_load**
**sp_mat_fetch**  
Reads one variables from a MAT-format file.

*(VAR *) `sp_mat_fetch(char *fn, char *vn, short *irc, short *nrc)`

**Parameters**
- **fn** file name
- **vn** variable name
- **irc** initial row and column
- **nrc** number of rows and columns

**Return Value**
- list of variables

**Remarks**
Elements of the variable list are of type VAR, which is described in Appendix A. The parameters irc and nrc specify a subset of the array. They each point to short arrays with two elements containing row and column values or may be set to NULL. Setting irc to NULL is equivalent to setting row and column to zero. Setting nrc to NULL is equivalent to setting rows and columns to the dimensions of the variable stored in the file.

**See also**
- `sp_mat_whos`, `sp_mat_save`
**sp_mat_load**
Reads all variables from a MAT-format file.

(VAR *) **sp_mat_load**(char *fn)

**Parameters**

- **fn** file name

**Return Value**
list of variables

**Remarks**
Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**

* sp_mat_whos, sp_mat_save
### sp_mat_save

Writes variables to a MAT-format file.

```c
(int) sp_mat_save(char *fn, VAR *vl)
```

#### Parameters
- **fn**: file name
- **vl**: variable list

#### Return Value
- error code, which is zero for no errors

#### Remarks
- Elements of the variable list are of type VAR, which is described in Appendix A.

#### See also
- `sp_mat_whos`, `sp_mat_load`
Signal Processing Library

*sp_mat_size*
Counts variables in a MAT-format file.

(int) *sp_mat_size*(char *fn)

**Parameters**

- fn file name

**Return Value**

number of variables

**Remarks**

Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**

*sp_mat_whos, sp_mat_save*
Signal Processing Library

*sp_mat_version*

Returns MAT file version number.

(int) sp_mat_version(char *fn)

**Parameters**

fn 
file name

**Return Value**

version number

**Remarks**

Version number is either 4 or 5 when the file is recognized as a valid MAT file. The version number is 0 when the file is not recognized as a valid MAT file.

**See also**

*sp_mat_size*
**Signal Processing Library**

\[
\text{sp\_mat\_whos}
\]

**sp\_mat\_whos**

Reads all variable names in a MAT-format file.

\[(\text{VAR *}) \ \text{sp\_mat\_whos}(\text{char *fn})\]

**Parameters**

- \( \text{fn} \) file name

**Return Value**

- list of variable names. This is the same as the variable list returned by \( \text{sp\_mat\_load} \), except without any data.

**Remarks**

Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**

\( \text{sp\_mat\_load, sp\_mat\_save} \)
Signal Processing Library

*sp_nxtpow2*
Returns the power of 2 that is greater than or equal to the input.

(int) sp_nxtpow2(int n)

**Parameters**

n array size

**Return Value**

none

**Remarks**

The returned value is a power of two that is greater than or equal to the input.
**sp_rand**
Generates uniform random values.

(void) sp_rand(float *x, int n)

**Parameters**
- x output array
- n array size

**Return Value**
- none

**Remarks**
- Generates n uniform random values between 0 and 1. Call sp_randseed to specify the generator seed before calling sp_rand.
**sp_randflat**
Generates random values with a flat spectrum

(int) `sp_randflat(float *x, int n)`

**Parameters**
- **x** output array
- **n** array size

**Return Value**
- 0 Success
- 1 N is not a power of 2

**Remarks**
Generates n random numbers with a flat spectrum. Call `sp_randseed` to specify the generator seed before calling `sp_randflat`. 
Signal Processing Library

**sp_randn**
Generates normal random values.

*(void)* `sp_randn(float *x, int n)`

**Parameters**
- **x** output array
- **n** array size

**Return Value**
none

**Remarks**
Uses the ziggurat method to generate `n` normally-distributed random values with mean 0 and standard deviation 1. Call `sp_randseed` to specify the generator seed before calling `sp_randn`. 
**Signal Processing Library**

**sp_randseed**  
Seeds the random number generator

*(void) sp_randseed(unsigned long s)*

**Parameters**

- **s**  
  seed

**Return Value**

- none

**Remarks**

Seeds the random number generator for sp_rand, sp_randflat, and sp_randn.
**sp_rcfft**
Performs in place real to complex fft.

(int) **sp_rcfft** (float *x, int n)

**Parameters**
- x 
  real input, complex output array
- n 
  input size + 2

**Return Value**
- 0 
  success

**Remarks**
Performs (real to complex) FFT on x (in place). If n is a power of two, then a fast Fourier transform is used; otherwise the Fourier transform is slow. The input x contains n real values. Returned are n/2+1 complex values (i.e. x[0]=real, x[1]=imaginary, etc). The x array size is n+2.

---

8 The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
**sp_sadd**
Scalar add

(void) **sp_sadd**(float *x, float *y, int n, double a)

**Parameters**
- x: input array
- y: output array
- n: array size
- a: scalar

**Return Value**
none

**Remarks**
Performs a scalar add: y = x+a.
Signal Processing Library

**sp_sma**
Scalar multiply and add

(void) **sp_sma**(float *x, float *y, int n, double b, double a)

**Parameters**

- **x**: input array
- **y**: output array
- **n**: array size
- **b**: scalar
- **a**: scalar

**Return Value**

none

**Remarks**

Performs scalar multiplication and addition: **y = x*b+a.**
**Signal Processing Library**

*sp_smul*
Scalar multiply

*(void) sp_smul(float *x, float *y, int n, double b)*

**Parameters**

- **x**: input array
- **y**: output array
- **n**: array size
- **b**: scalar

**Return Value**

none

**Remarks**

Performs scalar multiplication: \( y = x \times b \).
**sp_transfer**
Calculate transfer function given stimulus and response waveforms.

(int) `sp_transfer(float *x, float *y, int n, float *H)`

**Parameters**
- `x` stimulus waveform
- `y` response waveform
- `n` input array size
- `H` complex⁹ transfer function $H=\text{fft}(y)/\text{fft}(x)$

**Return Value**
Error code is zero when no error occurs.

**Remarks**
Computation is much faster when the input array size is a power of 2. The transfer function is complex valued, so real and imaginary components alternate. The transfer function will have $nf=(n/2+1)$ complex elements. The size of the transfer-function array should be equal to the input array size plus two.

**See Also**
- `sp_freqz`

---

⁹ The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
*sp_unwrap*
Unwrap phase.

**(void) sp_unwrap**(float *x, float *y, int n)

**Parameters**
- x  
  input phase array (cycles)
- y  
  output phase array (cycles)
- n  
  array size

**Return Value**
none

**Remarks**
Unwraps phase assuming that one cycle equals 1. The output array may be identical to the input array.
Signal Processing Library

*sp_vadd*
Vector add

( void ) *sp_vadd*( float *x, float *y, float *z, int n )

**Parameters**
- x input array
- y input array
- z output array
- n array size

**Return Value**
none

**Remarks**
- Adds two vectors:  \( z = x + y \).
Allocates memory for a list of variables.

**(VAR *)** _sp_var_alloc_(int _nvar_)

**Parameters**
- _nvar_  
  number of variables

**Return Value**
- list of (unspecified) variables.

**Remarks**
- Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**
- _sp_var_set_, _sp_mat_save_
**sp_var_clear**
Frees all memory for a list of variables.

( void ) sp_var_clear ( int nvar )

**Parameters**

- **nvar**
  number of variables

**Return Value**

none.

**Remarks**

Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**

- sp_var_set, sp_var_clear_all
Signal Processing Library

**sp_var_clear_all**
Frees all memory for all lists of variables.

(void) **sp_var_clear_all**

**Parameters**
none.

**Return Value**
none.

**Remarks**
Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**
*sp_var_set, sp_var_clear*
Signal Processing Library

\textit{sp\_var\_copy}
Copies a list of variables.

\begin{verbatim}
(VAR *) sp_var_clear(VAR *vl)
\end{verbatim}

\textbf{Parameters}
\begin{center}
\begin{tabular}{ll}
vl & list of variables \\
\end{tabular}
\end{center}

\textbf{Return Value}
list of variables.

\textbf{Remarks}
Elements of the variable list are of type VAR, which is described in Appendix A.

\textbf{See also}
\begin{verbatim}
sp\_var\_set, sp\_var\_allocate
\end{verbatim}
Signal Processing Library

**sp_var_find**
Find variable in list by name.

(int) sp_var_find(VAR *vl, char *vn)

**Parameters**
- **vl** list of variables
- **vn** variable name

**Return Value**
Variable index or -1 if not found.

**Remarks**
Searches variable list for specified name.

**See also**
*sp_mat_find*
Signal Processing Library

*sp_var_float*
Converting data type of all variables in a list to single-precision (32-bit) floating point.

(VOID) **sp_var_float**(VAR **vl**)

**Parameters**

**vl**
list of variables

**Return Value**

none.

**Remarks**

Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**

*sp_var_set, sp_var_allocate*
Signal Processing Library

**sp_var_f4**
Returns one single-precision float value from variable list by name.

(float) `sp_var_f4(VAR *vl, char *vn)`

**Parameters**
- `vl` list of variables
- `vn` variable name

**Return Value**
First value in variable array.

**Remarks**
Searches variable list for specified name.

**See also**
- `sp_var_f8`, `sp_var_i2`, `sp_var_i4`
Signal Processing Library

*sp_var_f8*

Returns one double-precision float value from variable list by name.

(double)  

```
sp_var_f8(VAR *vl, char *vn)
```

Parameters

- `vl` list of variables
- `vn` variable name

Return Value

First value in variable array.

Remarks

Searches variable list for specified name.

See also

- `sp_var_f4`, `sp_var_i2`, `sp_var_i4`
Signal Processing Library

*sp_var_i2*
Returns one short-integer value from variable list by name.

*(short) sp_var_i2(VAR *vl, char *vn)*

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>vl</td>
<td>list of variables</td>
</tr>
<tr>
<td>vn</td>
<td>variable name</td>
</tr>
</tbody>
</table>

**Return Value**
First value in variable array.

**Remarks**
Searches variable list for specified name.

**See also**

*sp_var_f4, sp_var_f8, sp_var_i4*
**sp_var_i4**
Returns one long-integer value from variable list by name.

`(long) sp_var_i4(VAR *vl, char *vn)`

**Parameters**
- `vl` : list of variables
- `vn` : variable name

**Return Value**
First value in variable array.

**Remarks**
Searches variable list for specified name.

**See also**
- `sp_var_f4`, `sp_var_f8`, `sp_var_i2`
**sp_var_set**
Specifies variable properties in a list of variables.

(void) **sp_var_set**(VAR *vl, char *name, void *data, int rows, int cols, char *frmt)

**Parameters**
- **vl**: pointer to a variable in a list of variables
- **name**: variable name
- **data**: pointer to data array to be assigned to this variable
- **rows**: number of rows
- **cols**: number of column
- **frmt**: string of characters specifying data type

**Return Value**
- none.

**Remarks**
Elements of the variable list are of type VAR, which is described in Appendix A. The data format string should begin with I, U, F, or T to specify integer, unsigned integer, floating-point, or text, respectively. When the first letter is I or U, it should be followed by 1, 2, or 4 to specify the number of bytes of integer precision. When the first letter is F, it should be followed by 4 or 8 to specify the number of bytes of floating-point precision. The number in the format string may also be followed by C to specify complex\(^{10}\) data.

**See also**
- **sp_var_set, sp_var_allocate**

---

\(^{10}\) The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.
**sp_var_size**
Count variables in a list of variables.

*(void) sp_var_size(VAR *vl)*

**Parameters**
- **vl**
  pointer to a variable in a list of variables

**Return Value**
number of variables.

**Remarks**
Elements of the variable list are of type VAR, which is described in Appendix A.

**See also**
- `sp_var_set`, `sp_var_allocate`
Signal Processing Library

\emph{sp\_version}

Returns SigPro version string.

(char *) \textbf{sp\_version}()

\textbf{Parameters}

none

\textbf{Return Value}

version string

\textbf{Remarks}

For example, "SigPro version 0.05, 13-Dec-05".
**sp_vdot**

Returns vector dot product.

\[
\text{(double) } \text{sp_vdot}(\text{float } x, \text{float } y, \text{int } n)
\]

**Parameters**

- **x**: input array
- **y**: input array
- **n**: array size

**Return Value**

Vector dot product

**Remarks**

Computes the dot product of vectors \(x\) and \(y\).
Signal Processing Library

**sp_vdiv**
Vector divide

(int) **sp_vdiv**(float *x, float *y, float *z, int n)

**Parameters**
- **x**  input array
- **y**  input array
- **z**  output array
- **n**  array size

**Return Value**
Number of divisions by zero (not performed). Values are returned for all non-zero divisions.

**Remarks**
Divides two vectors: **z = x/y**.
Signal Processing Library

*sp_vmax*

Vector maximum

(int) **sp_vmax**(float *x, int n)

**Parameters**

- **x**  
  input array
- **n**  
  array size

**Return Value**

Index of first element with maximum value
Signal Processing Library

\textit{sp_vmin}
Vector minimum

(int) \textbf{sp_vmin}(float *x, \textbf{int n})

\textbf{Parameters}
\begin{itemize}
\item \textbf{x} \hspace{1cm} input array
\item \textbf{n} \hspace{1cm} array size
\end{itemize}

\textbf{Return Value}
Index of first element with minimum value
Signal Processing Library

\textit{sp_vmul}

Vector multiply

(\textbf{void}) \textbf{sp_vmul}(float *x, float *y, float *z, int n)

\textbf{Parameters}
\begin{itemize}
  \item \textbf{x} \hspace{1cm} \text{input array}
  \item \textbf{y} \hspace{1cm} \text{input array}
  \item \textbf{z} \hspace{1cm} \text{output array}
  \item \textbf{n} \hspace{1cm} \text{array size}
\end{itemize}

\textbf{Return Value}
\begin{itemize}
  \item none
\end{itemize}

\textbf{Remarks}
Multiplies each element of two vectors: \[ z = x\cdot y. \]
Signal Processing Library

*sp_vsub*
Vector subtract

(VOID) *sp_vsub*(float *x, float *y, float *z, int n)

**Parameters**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>input array</td>
</tr>
<tr>
<td>y</td>
<td>input array</td>
</tr>
<tr>
<td>z</td>
<td>output array</td>
</tr>
<tr>
<td>n</td>
<td>array size</td>
</tr>
</tbody>
</table>

**Return Value**

none

**Remarks**

Subtracts two vectors: \( z = x - y \).
Signal Processing Library

\textit{sp\\_wav\\_info}

Read waveform information from WAV file.

\texttt{(VAR *) sp\\_wav\\_info(char *fn, float *fs)}

**Parameters**

\begin{align*}
\text{fn} & \quad \text{file name} \\
\text{fs} & \quad \text{sampling rate (samples/sec)}
\end{align*}

**Return Value**

Variable containing waveform information, but not the waveform.

**Remarks**

The number of samples in the waveform is the number of rows in the VAR structure. The number of channels in the waveform is the number of cols in the VAR structure.

**See also**

\textit{sp\\_wav\\_read}
signal Processing Library

*sp_wav_read*
Read waveform from WAV file.

(VAR *) **sp_wav_read**(char *fn, int *ifr, int *nfr, float *fs)

**Parameters**
- **fn**  
  file name
- **ifr**  
  pointer to initial frame
- **nfr**  
  pointer to number of frames
- **fs**  
  sampling rate (samples/sec)

**Return Value**
Variable containing waveform, possibly with multiple channels.

**Remarks**
The number of samples in the waveform is the number of rows in the VAR structure. The number of channels in the waveform is the number of cols in the VAR structure. The waveform data type is float. Partial reads are possible by specifying the initial frame and number of frames. A frame includes all columns of a single row and corresponds with all channels for a single sample time. When the ifr is NULL the first frame is the initial frame. When the nfr is NULL all samples are read from the initial frame to the end of the file.

**See also**
*sp_wav_info*
**sp_window**

Standard window

(int) **sp_window**(float *, int n, int wt)

**Parameters**

- **y**  
  output array
- **n**  
  array size
- **wt**  
  window type

**Return Value**

- 0  
  Success
- 1  
  invalid window type

**Remarks**

The window types are 0=rectangular (ones), 1=triangular (Bartlet), 2=Hanning, 3=Hamming, 4=Blackman, 5=Nuttall.
Signal Processing Library

*sp_zero*
Zeros an array

(VOID) **sp_zero**(float *y, int n)

**Parameters**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>y</strong></td>
<td>output array</td>
</tr>
<tr>
<td><strong>n</strong></td>
<td>array size</td>
</tr>
</tbody>
</table>

**Return Value**

none

**Remarks**

Sets all values in array y to 0.
Appendix A. MAT and VAR functions

The functions that load variables from MAT files and save variables to MAT files make use of variables lists that are VAR arrays. The VAR struct is defined in sigpro.h.

```c
struct {
    char *name;
    void *data;
    long rows, cols;
    char dtyp, cmpx, text, last;
}
```

An empty VAR list is created by calling `sp_var_alloc`. The last element in a variable list is indicated by setting the `last=1`. Variable properties may be specified by calling `sp_var_set`. Memory allocated to a single variable list may be freed by calling `sp_var_clear`. Memory allocated to all variable lists may be freed by calling `sp_var_clear_all`. A variable list may be copied by calling `sp_var_copy`. All data in variable list may be converted to single-precision floating point by calling `sp_var_float`. The `sp_var_size` function simply counts the number of variables in a variable list.

Four function support MAT files. The `sp_mat_save` function creates version 4 MAT files. The `sp_mat_load` function reads either version 4 or version 5 MAT files. The `sp_mat_whos` function is similar to the `sp_mat_load` function, except that the data is omitted from the variable list. This is useful when only the variable properties are of interest. The `sp_mat_size` function simply counts the number of variables in a MAT file.

The data type for `rows` and `cols` was changed from short to long in version 0.22.
Appendix B. OPT structure

This structure provides options that allow control over the iteration performed by the `sp_fmins` function. The OPT struct is defined in sigpro.h.

```
struct {
    float icons, ifrac;
    float tolfun, tolx;
    int display, funchk;
    int maxeval, maxiter, miniter;
    int (*escape)(void);
    void (*report)(float *);
}
```

These variables are described below and default values are given in brackets.

- **icons** – Constant used to offset zeros in the initial parameter list when creating a starting simplex. [0.00025]
- **ifrac** – Fraction to offset non-zero values in the initial parameter list when creating a starting simplex. [0.05]
- **ffrac** – Minimum change required in successive parameter with the largest fractional change to allow iteration to continue. [0.0001]
- **tolfun** – Minimum change required in successive error function values to allow iteration to continue. [Not implemented.]
- **tolx** – Minimum change required in successive parameter-list norms to allow iteration to continue. [Not implemented.]
- **display** - [Not implemented.]
- **funchk** - [Not implemented.]
- **maxeval** - [Not implemented.]
- **maxiter** – Maximum number of iterations. [1000]
- **miniter** – Minimum number of iterations. [Number of parameters.]
- **escape** – Callback function terminates iteration when it returns true. [Null]
- **report** – Callback function allows intermediate parameter values to be printed. [Null]
Appendix C. Test Programs

Several programs are included in the source code distribution that test some of the features of the SIGPRO library.

- `tst_afd` – Test analog filter design.
- `tst_fft` – Test real & complex\(^{11}\) FFT and inverse FFT.
- `tst_mat` – Test MAT file save & load.
- `tst_min` – Test fmins minimization function.
- `tst_shp` – Test frequency-shaping functions.
- `tst_src` – Test sampling-rate conversion.
- `tst_wav` – Test WAV file read & write.
- `tst_xfr` – Test transfer-function computation.

\(^{11}\) The old-style complex format is used in which real and imaginary components alternate within a single float array with size equal to twice the number of complex values.