SPro

Speech Signal Processing Toolkit, release 5.0.
Last updated 9 November 2010.

Guillaume Gravier

Table of Contents

1 Introduction

1.1 What is SPro?

SPro is a speech signal processing toolkit which provides runtime commands implementing standard feature extraction algorithms for speech and speaker recognition applications and a C library to implement new algorithms and to use SPro files within your own programs.

SPro was originally designed for variable resolution spectral analysis but also provides for feature extraction techniques classically used in speech applications. There are commands for the following representations:

- filter-bank energies
- cepstral coefficients (filter-bank or linear prediction)
- linear prediction derived representation (prediction and reflection coefficients, log area ratios and line spectrum pairs)

Though the toolkit has been designed as a front-end for applications such as speech or speaker recognition, we believe the library provides enough possibilities to implement various feature extraction algorithms easily (e.g. zero crossing rate). However, no command for such features is provided.

The library, written in ANSI C, provides functions for the following:

- waveform signal input
- low-level signal processing (FFT, LPC analysis, etc.)
- low-level feature processing (lifter, CMS, variance normalization, deltas, etc.)
- feature I/O

The library does not provide for high-level feature extraction functions which directly converts a waveform into features, mainly because such functions would require a tremendous number of arguments in order to be versatile. However, it is rather trivial to write such a function for your particular needs using the SPro library.

1.2 How to read this manual?

The manual is divided into three main parts:

1. user manual
2. programmer manual
3. reference manual

Chapter 3 [SPro tools], page 11 is the user manual. It provides a description of the speech analysis algorithms involved (see Chapter 2 [Speech analysis], page 5) and explains in details the use and the implementation of the SPro commands `sfbank`, `sfbcep`, `slpc`, `slpcep` and `scopy`. Section 3.1 [File formats], page 11 describes the supported waveform file formats and the SPro feature file format. The next sections are dedicated to the detailed description of the SPro tools.

Chapter 4 [SPro library], page 23 is the programmer manual which describes the library main data structures and the associated functions.

Chapter 5 [Reference guide], page 41 provides a quick reference manual for the SPro tools syntax.

If you have been using a former version of SPro, read Section 6.3 [Compatibility], page 55 carefully for crucial information on the (in)compatibility of SPro 5.0 with the previous versions.

Finally, to learn more about the evolution of SPro, the history of the various SPro releases is detailed in Chapter 6 [Changes], page 55.

1.3 Installing SPro

Installation follows the standard GNU installation procedure. The two following lines in your favorite shell

```
./configure
make
```

will build the library and the runtimes. SPro supports some extra features based on some external packages. These features can be turned on/off (depending on whether you have them already installed on your machine) using the '`--with-xxx`' options of the configure script. Supported enable options are:

```
--with-sphere[=path]     SPHERE 2.6 file format support
```

If the SPHERE library is installed in a standard place on your system (e.g. '`/usr/local/include`' and '`/usr/local/lib`'), there is no need to specify *path*. Otherwise, *path* should point to the directory where the SPHERE library has been installed. `configure` will search for the library includes in *path*`/include` and for the archives in *path*`/lib`. Compiling SPro with the '`-O3`' option of the `gcc` compiler (`CFLAGS=-O3`) is a good idea for sake of rapidity.

Before installing, you may want to check your build by typing

```
make check
```

Finally, installing the library, the runtimes and the info documentation can be done running

```
make install
```

The installation path is specified by the configuration script (try `./configure --help` for details) and defaults to '`/usr/local`'.

See file '`INSTALL`' in the distribution top directory for more details.

To the author knowledge, SPro has been successfully build and used on Linux, SPARC/SunOS, and HP-UX. It should also work on AIX though this has not been tested so far.

1.4 License

As of release 5.0, SPro is distributed as a free software under the MIT License agreement:

> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in

Previous releases are distributed under the GNU Public License agreement.

1.5 Reporting bugs

Bugs should be reported to `guig@irisa.fr`. Feel free to submit a diagnostic or even a patch along with your bug report if you kindly bothered to do the trouble-shooting. This is always appreciated.

1.6 Contributors

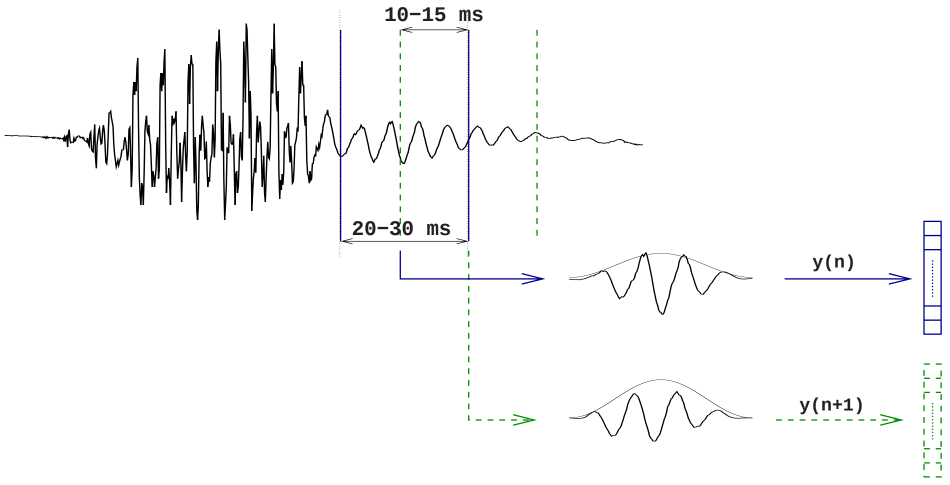
Along the years, SPro has benefited from the help of several contributors. Here is a list, in alphabetical order, of those brave souls who contributed code to the software: Raphaël Blouet, Pierre Duhamel, Sacha Krstulovic, Johnny Mariethoz, Sylvain Meigner, Alexey Ozerov, Jacques Prado and Frederic Wils.

2 Speech analysis techniques

This section provides a brief scientific overview of the speech signal analysis techniques involved in SPro with a particular focus on variable resolution spectral analysis. It also defines the equations and methods implemented in SPro.

2.1 Pre-emphasis and windowing

Speech is intrinsically a highly non-stationary signal. Therefore, speech analysis, whether FFT-based or LPC-based, must be carried out on short segments across which the speech signal is assumed to be stationary. Typically, the feature extraction is performed on 20 to 30 ms windows with 10 to 15 ms shift between two consecutive windows. This principle is illustrated in the figure below



To avoid problems due to the truncation of the signal, a weighting window with the appropriate spectral properties must be applied to the analyzed chunk of signal. SPro implements three such windows

HAMMING	$w_i = 0.54 - 0.46\cos(i\boxed{\pi^2}/N) \longrightarrow 2\pi$
HANNING	$w_i = (1 - \cos(i\boxed{\pi^2}/N))/2$
BLACKMAN	$w_i = 0.42 - 0.5\cos(i\boxed{\pi^2}/N) + 0.08cos(2i\boxed{\pi^2}/N)$

where N is the number of samples in the window and $i \in [0, N-1]$.

Pre-emphasis is also traditionally use to compensate for the -6dB/octave spectral slope of the speech signal. This step consists in filtering the signal with a first-order high-pass filter $H(z) = 1 - kz^{-1}$, with $k \in [0, 1[$. The pre-emphasis filter is applied on the input signal before windowing.

2.2 Variable resolution spectral analysis

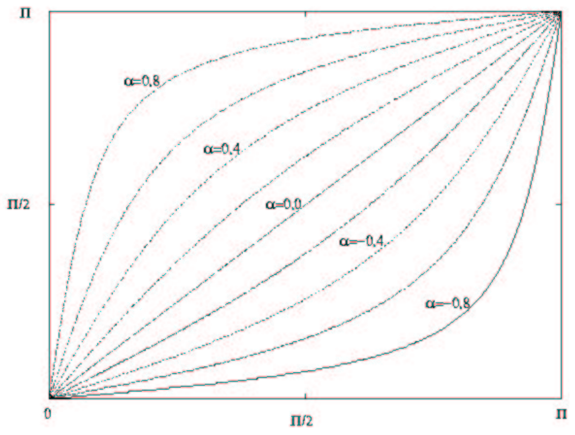
Classical spectral analysis has a constant resolution over the frequency axis. The idea of variable resolution spectral analysis[1] is to vary the spectral resolution as a function of

[1] Variable resolution spectral analysis of a signal is presented in details in *C. Chouzenoux, Analyse spectrale à résolution variable: application au signal de parole, Ph.D. thesis, ENST Paris, 1982*, where it is applied to speech coding.

the frequency. This is achieved by applying a bilinear transformation of the frequency axis, the transformation being controlled by a single parameter a. The bilinear warping of the frequency axis is defined by

$$f' = \arctan \left| \frac{(1-a^2)\sin f}{(1+a^2)\cos f - 2a} \right|$$

where f and f' are the frequencies on the original and transformed axis respectively and $a \in]-1,1[$. The axis transformation is depicted in the following figure



Spectral analysis is done with a constant resolution on the warped axis f' and therefore with a variable resolution on the original axis. Clearly, positive values of a leads to a higher low frequency resolution while negative values give a better high frequency resolution. If a equals one, the transformation is the identity thus resulting in a classical constant resolution spectral analysis.

Using variable resolution spectral analysis with a filter-bank is rather trivial since it simply consists in determining the filter's central frequency according to the warping. See Section 2.3 [Filter-banks], page 7.

Linear predictive models with variable resolution spectral analysis is also possible. Very briefly, the idea consists in solving the normal equations on the *generalized* auto-correlation rather than on the traditional auto-correlation sequence. The generalized auto-correlation $r(p)$ is the correlation between the original signal filtered by a corrective filter

$$\mu(z) = \frac{1-a^2}{(1-az^{-1})^2}$$

and the latter filtered p times by a correction filter of response

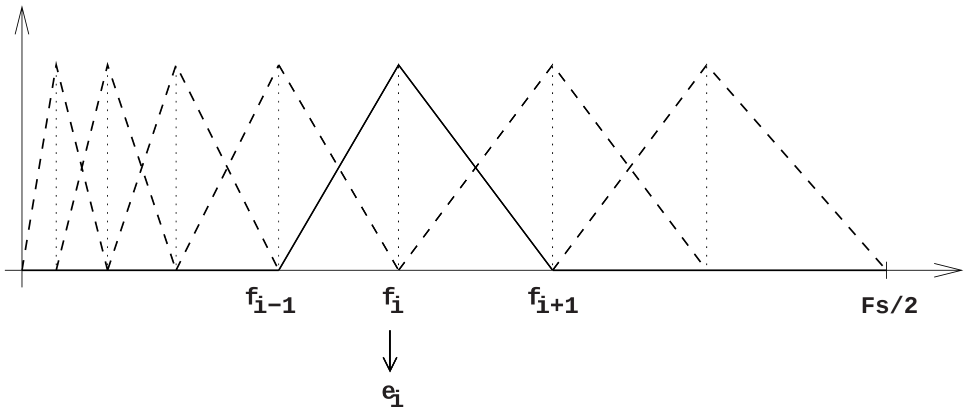
$$H(z) = \frac{z^{-1} - a}{1 - az^{-1}}$$

See Section 2.4 [LPC analysis], page 8, for more details.

2.3 Filter-bank analysis

Filter-bank is a classical spectral analysis technique which consists in representing the signal spectrum by the log-energies at the output of a filter-bank, where the filters are overlapping band-pass filters spread along the frequency axis. This representation gives a rough approximation of the signal spectral shape while smoothing out the harmonic structure if any. When using variable resolution analysis, the central frequencies of the filters are determined so as to be evenly spread on the warped axis and all filters share the same bandwidth on the warped axis. This is also applied to Mel frequency warping, a very popular warping in speech analysis which mimics the spectral resolution of the human ear. The Mel warping is approximated by $mel(f) = 2595 \log_{10}(1 + f/700)$.

SPro provides an implementation of filter-bank analysis with triangular filters on the FFT module as depicted below



The energy at the output of channel i is given by

$$e_i = \log \sum_{j=1}^{N} h_i(j) \ \ ||X(j)||$$

where N is the FFT length[2] and h_i is the filter's frequency response as depicted above. The filter's response is a triangle centered at frequency f_i with bandwidth $[f_{i-1}, f_{i+1}]$, assuming the f_i's are the central frequencies of the filters determined according to the desired spectral warping.

[2] Actually half of the FFT length.

2.4 Linear predictive analysis

Linear prediction is a popular speech coding analysis method which relies on a source/filter model if the speech production process. The vocal tract is modeled by an all-pole filter of order p whose response is given by

$$H(z) = \frac{1}{1 + \sum_{i=1}^{p} a_i z^{-i}} \quad .$$

The coefficients a_i are the prediction coefficients, obtained by minimizing the mean square prediction error. The minimization is implemented in SPro using the *auto-correlation* method.

The idea of the resolution algorithm is to iteratively estimate the prediction coefficients for each prediction order until the required order is reached. Assuming the prediction coefficients for order $n-1$ are known and yields a prediction error e_{n-1}, the estimation of the coefficients for order n rely on the n'th reflection coefficients defined as

$$k_n = \frac{-1}{e_{n-1}} \sum_{i=0}^{n-1} a_{n-1}(i) r(n-i) \ ,$$

where r is the autocorrelation of the signal. Given the reflection coefficient k_n, the prediction coefficients are obtained using the recursion

$$a_n(i) = a_{n-1}(i) + k_n a_{n-1}(n-i)$$

for $i = 1, \ldots, n-1$ and $a_n(n) = k_n$. Finally, the prediction error for order n is given by

$$e_n = e_{n-1}(1 - k_n^2) \ .$$

For variable resolution, the *generalized* auto-correlation sequence is used instead of the traditional auto-correlation. See Section 2.2 [Variable resolution], page 5. for details on generalized auto-correlation.

The all-pole filter coefficients can be represented in several equivalent ways. First, the linear prediction coefficients a_i can be used directly. The reflection (or partial correlation) coefficients $k_i \in]-1, 1[$ used in the resolution algorithm can also be used to represent the filter. The log-area ratio, defined as

$$g_i = 10 \log_{10} \left(\frac{1 + k_i}{1 - k_i} \right) \ ,$$

is also a popular way to define the prediction filter. Last, the line spectrum frequencies (a.k.a. line spectrum pairs) are also frequently used in speech coding. Line spectrum frequencies is another representation derived from linear predictive analysis which is very popular in speech coding.

2.5 PLP analysis

Perceptual Linear Prediction (PLP) is combines filter-bank analysis and linear prediction to compute linear prediction coefficients on a perceptual spectrum. The filter-bank power spectrum is filtered using an equal loudness curve and passed through a compression function $f(x) = x^{1/n}$ where usually n=3, thus resulting in an auditory spectrum from which the autocorrelation is computed by inverse discrete Fourier transform. Linear prediction coefficients are then carried out as usual from the autocorrelation.

2.6 Cepstral analysis

Probably the most popular features for speech recognition, the cepstral coefficients can be derived both from the filter-bank and linear predictive analyses. From the theoretical point of view, the cepstrum is defined as the inverse Fourier transform of the logarithm of the Fourier transform module. Therefore, by keeping only the first few cepstral coefficients and setting the remaining coefficients to zero, it is possible to smooth the harmonic structure of the spectrum[3]. Cepstral coefficients are therefore very convenient coefficients to represent the speech spectral envelope.

In practice, cepstral coefficients can be obtained from the filter-bank energies e_i via a discrete cosine transform (DCT) given by

$$c_i = \sqrt{\frac{2}{N}} \sum_{j=1}^{N} e_j \cos\left(\frac{\pi\, i\, (j-0.5)}{N}\right) \quad ,$$

where N is the number of channels in the filter-bank and $i \in [1, M]$ $(M <= N)$. Cepstral coefficients can also be obtained from the linear prediction coefficients a_i according to

$$c_i = -a_i + \frac{1}{i} \sum_{j=1}^{i-1} (i-j)\, a_j\, c_{i-j} \quad ,$$

for $i \in [1, M]$ with $M <= P$, the prediction order.

Cepstral coefficients have rather different dynamics, the higher coefficients showing the smallest variances. It may sometimes be desirable to have a constant dynamic across coefficients for modeling purposes. One way to reduce these differences is liftering which consists in applying a weight to each coefficients. The weight for the i'th coefficient is defined in a parametric way according to

$$h_i = 1 + \frac{L}{2} \sin\left(\frac{i\,\pi}{L}\right) \quad ,$$

where L is the lifter parameter, typically equals to $2M$.

2.7 Deltas and normalization

Feature normalization can be used to reduce the mismatch between signals recorded in different conditions. In SPro, normalization consists in mean removal and eventually variance normalization. Cepstral mean subtraction (CMS) is probably the most popular compensation technique for convolutive distortions. In addition, variance normalization consists in normalizing the feature variance to one and is a rather popular technique in speaker recognition to deal with noises and channel mismatch. Normalization can be global or local. In the first case, the mean and standard deviation are computed globally while in the second case, they are computed on a window centered around the current time.

To account for the dynamic nature of speech, it is possible to append the first and second order derivatives of the chosen features to the original feature vector. In SPro, the first order

[3] Somehow, zeroing the last cepstral coefficients is like applying a low-pass filter to the (log module of) the original signal spectrum.

derivative of a feature y_i is approximated using a second order limited development given by

$$y_i'(t) = \frac{2\ y_i(t+2) + y_i(t+1) - y_i(t-1) - 2\ y_i(t-2))}{10} \quad .$$

Second order differences, known as accelerations, are obtained by derivating the first order differences. It is therefore not possible to have the acceleration without the delta features.

3 The SPro tools

3.1 File formats

This section describes the file formats manipulated by SPro. Most SPro tools input signal from a *waveform stream* and output feature vectors to a *feature stream*.

3.1.1 Waveform streams

Waveform streams are files which contains the signal samples, either in raw PCM format or in an encoded format to save disk space. Currently, SPro supports raw, mono, 16bits/sample files, as well as A-law or U-law compressed 8bits/sample files[1], WAVE files and optionally SPHERE[2] files. The SPHERE format is only supported if SPro has been compiled with the SPHERE library (‘`--with-sphere`’ in `configure`). Raw format (i.e. with no header) with a 8 kHz sample rate is the default assumed by SPro if not otherwise specified.

Waveform are considered as streams by SPro and are read via an input buffer which means they can be of arbitrary (even infinite) length. Even file formats for which the number of samples is known in advance from the header will not be entirely loaded into memory. In particular, this mechanism makes it possible to read waveforms from the standard input even though the number of signals is not known offhand. One particularly interesting consequence is the possibility to pipe the output of an external command into the input of a SPro command. For example, it is possible using a pipe to support file formats which are not supported by SPro. The following line

```
madplay --left --output=raw:- foo.mp3 | sfbcep -f 11025 - foo.mfcc
```

shows how to decode the left channel of an MP3 encoded file (‘`foo.mp3`’) into a raw, mono, 16 bits/sample file which is then piped into the `sfbcep` tool, assuming the sample rate of the MP3 file is 11,025 Hz.

3.1.2 Feature streams

A feature streams is a file containing feature vectors. The format used to store the feature vectors is specific to SPro and consists of a header followed by data. The header itself is divided in two parts, an optional variable length header and a fixed length compulsory header.

To avoid byte-order problems, binary parts of the feature streams, such as the fixed length header and the feature vectors, are always stored in little-endian format (Intel-like processor) and therefore must be swapped if read on a big-endian (Motorola-like processor) machine. Byte swapping is automatically taken care of when using the library functions to read SPro streams. See Chapter 4 [SPro library], page 23, for details on SPro stream I/O functions.

[1] U-law and A-law formats are used in telephony, or alternately as the standard audio format in the SUN SOLARIS OS.

[2] SPHERE is the file format used by most NIST tools and databases. See `http://www.nist.gov/speech` for the SPHERE package.

The variable length header is an optional ASCII header containing '`attribute = value`' statements, starting with a '`<header>`' tag and ending with '`</header>`'. The following is a sample variable length header:

```
<header>
a_field = an arbitrary value;            # a comment

date = Wed Jul 23 14:59:12 CEST 2003;    # this is the date
snr = 20 dB;                             # SNR
</header>
```

Both the '`attribute`' and '`value`' strings are arbitrary. Note that as of now, none of the SPro tools output variable length headers. However, such headers are supported and can be added using the `cat` or `bcat` command. For example, the command

```
bcat header.txt foo.mfcc > bar.mfcc
```

could be used to add the variable length header contained in file '`header.txt`' to the output of an SPro command '`foo.prm`', the resulting file being '`bar.prm`'. The header file '`header.txt`' is a regular text file containing text such as given in the example above, where the last line of the file must consist of the '`</header>`' tag, possibly with a carriage return.

The compulsory fixed length header is a 10 byte binary header containing the feature vector dimension[3] (`unsigned short` = 2 bytes), a flag describing the content of the feature vector (`long` = 4 bytes) and the frame rate in Hz (`float` = 4 bytes). The feature stream description flag is actually a field of bits with the following meaning

bit	**letter**	**description**
1	'`E`'	feature vector contains log-energy.
2	'`Z`'	mean has been removed
3	'`N`'	static log-energy has been suppressed (always with '`E`' and '`D`')
4	'`D`'	feature vector contains delta coefficients
5	'`A`'	feature vector contains delta-delta coefficients (always with '`D`')
6	'`R`'	variance has been normalized (always with '`Z`')

The letter in the second column corresponds to the letter used in all the SPro tools to modify or visualize the feature description flags.

Feature vectors, or data, are stored after the header in time ascending order. A feature vector is a binary vector of `float`'s as illustrated in the following example

```
+-----------------+---+-----------------+----+-----------------+---+
|     static      | E |      delta      | dE |   delta delta   |ddE|
+-----------------+---+-----------------+----+-----------------+---+
```

with the static coefficient first, optionally followed by the log-energy, the delta and delta-delta features as indicated by the feature description flag.

[3] Note that, as opposed to previous versions if SPro, the dimension in the header correspond to the total feature vector dimension.

3.2 Common options

Here is a list of options common to all (or most of) the tools. The `scopy` feature manipulation tool options slightly differ from the list below since most of the options are concerned with waveform processing.

3.2.1 I/O options

The following options are used to control the waveform and feature I/Os:

`-F, --format=str`
: Specify the input waveform file format. The format string `str` is one of '`PCM16`', '`ALAW`', '`ULAW`', '`wave`' or '`sphere`', the latter being possible only if SPro was compiled with the SPHERE library. Argument is case insensitive. Default value is '`PCM16`'.

`-f, --sample-rate=f`
: Set input waveform sample rate to `f` Hz for '`PCM16`', '`ALAW`' and '`ULAW`' waveform files. This option is ignored for waveform file formats for which the sample rate is specified in the header. Default value is 8,000 Hz.

`-x, --channel=n`
: For multiple channel waveform files, set the channel to consider for feature extraction. Default value is 1.

`-B, --swap`
: Swap the input waveform samples. This is particularly useful for waveform files generated on a machine with a different endian. Default is not to swap.

`-I, --input-bufsize=n`
: Set the input buffer size to `n` kbytes. The smaller the input buffer size, the more disk access and therefore, the slower the program is. So you will have to choose between speed and memory! Default is 10 Mbytes.

`-O, --output-bufsize=n`
: Set the output buffer size to `n` kbytes. Again, you need a compromise between speed and memory requirements. However, one important point is that global processing such as mean subtraction, energy normalization and delta computation are done on the buffer basis (i.e. such processings are done only when the buffer is full or at the end of the stream, whichever comes first) which introduces some inconsistencies at the buffer boundaries[4]. Using a small output buffer size can then result in many boundary problems and it is recommended not to diminish the output buffer size below a couple of thousand frames. Default is 10 Mbytes.

`-H, --header`
: Output extended (variable length) header in addition to the mandatory header.

[4] This is a known 'bug' that should be corrected someday. It is actually rather impossible to correct the bug for global normalization which would require to store all of the data into memory. However, it is possible — and probably desirable — to correct things when a sliding window is specified.

3.2.2 Waveform framing options

Waveform framing is driven by the following options:

`-k, --pre-emphasis=f`
: Set the pre-emphasis coefficient to `f`. Default is 0.95.

`-l --length=f`
: Set the analysis frame length to `f` ms. Default is 20.0 ms.

`-d, --shift=f`
: Set the interval between two consecutive frames to `f` ms. Default is 10.0 ms.

`-w, --window=str`
: Specify the waveform weighting window. The window is one of '`Hamming`', '`Hanning`', '`Blackman`' or '`none`'. If the argument is '`none`', no window is applied. Argument is case insensitive. Default is '`Hamming`'.

3.2.3 Feature vector options

The following options are used to control the content of the output feature vectors, enabling global normalizations and dynamic feature computation:

`-Z, --cms`
: Perform mean normalization.

`-R, --normalize`
: Perform variance normalization. Variance normalization is only possible if '`--cms`' is also specified. Otherwise, an error is generated.

`-L, --segment-length=n`
: Set normalization and energy scaling segment length. If this option is specified, mean, variance or max calculation is performed using a sliding window of '`n`' frames. Default is to calculate mean, variance or max globally when flushing the output buffer. This argument is ignored if neither '`--cms`' nor '`--normalize`' are specified.

`-D, --delta`
: Add first order derivatives to the feature vector.

`-A, --acceleration`
: Add second order derivatives to the feature vector. This is only possible if '`--delta`' is also specified. Otherwise, an error is generated.

`-N, --no-static-energy`
: Remove static log-energy from the feature vector. This is only possible if '`--delta`' is also specified. Otherwise, an error is generated.

3.2.4 Miscellaneous options

Last but not least, here are some very practical options (specially the second one):

`-v, --verbose`
: Turn on verbose mode

`-h, --help`
: Print a help message for the tool and exit.

`-V, --version`
: Print version information and exit.

3.3 I/O via stdin and stdout

Every SPro command requires that input and output files are explicitly specified. However, in the very Unix philosophy, the special symbol '`-`' (dash) can be used as input file to specify that input is to be read from `stdin` or as output file to specify that output should be directed to `stdout`.

The use of standard input and output makes it possible to pipe the SPro commands one after the other or even with external programs. The example

```
sfbcep foo.lin - |  scopy -o ascii - -
```

illustrates the use of pipes to list the feature vectors directly from the waveform file '`foo.lin`'. Another particularly useful example of pipes with SPro commands is given in Section 4.1 [Waveform streams], page 23.

3.4 Extracting features

3.4.1 Filter-bank analysis tools

The tools `sfbank` and `sfbcep` are dedicated to filter-bank based speech analysis.

Filter-bank log-magnitude features

The first filter-bank analysis tool, `sfbank`, takes as input a waveform and output filter-bank magnitude features. For each frame, the FFT is performed on the windowed signal, possibly after zero padding, and the magnitude is computed before being integrated using a triangular filter-bank. See Section 2.3 [Filter-banks], page 7, for mathematical details. To avoid numerical problems, a threshold is used to keep channel log-magnitudes positive or null. The signal bandwidth may be artificially limited by specifying lower and higher frequencies using the '`--freq-min`' and '`--freq-max`' options respectively. In this case, the central frequencies of the filter-bank channels are regularly taken in the specified bandwidth. Even if frequency warping is used, the lower and upper frequencies are specified in the linear frequency domain, though, of course, the filter's central frequencies will be taken regularly in the transformed domain. Both Mel and bilinear frequency warping are possible with `sfbank`.

First and second order derivatives can be appended to the filter-bank log-magnitude features using '`--delta`' and '`--acceleration`' respectively.

Filter-bank cepstral features

The second filter-bank analysis tool, `sfbcep`, takes as input a waveform and output filter-bank derived cepstral features. The filter-bank processing is similar to what is done in `sfbank` (see previous section). The cepstral coefficients are computed by DCT'ing the filter-bank log-magnitudes and possibly liftered.

Optionally, the log-energy can be added to the feature vector. In `sfbcep`, the frame energy is calculated as the sum of the squared waveform samples after windowing. As for the magnitudes in the filter-bank, the log-energy are thresholded to keep them positive or null. The log-energies may be scaled to avoid differences between recordings.

Mean and variance normalization of the static cepstral coefficients can be specified with the global '`--cms`' and '`--normalize`' options but do not apply to log-energies. The normalizations can be global (default) or based on a sliding window whose length is specified with '`--segment-length`'.

Finally, first and second order derivatives of the cepstral coefficients and of the log-energies can be appended to the feature vectors. When using delta features, the absolute log-energy can be suppressed using the '`--no-static-energy`' option.

Options

The following options are available for both `sfbank` and `sfbcep`.

`-n, --num-filters=n`
: Specify the number of channels in the filter bank. Default is 24.

`-a, --alpha=f`
: Use bilinear frequency warping and set the warping parameter a to `f` (`f` must be between 0 and 1). This option is incompatible with '`--mel`' and will be overwritten by the latter. Default is no warping.

`-m, --mel`
: Use Mel frequency warping. This option overwrites the '`--alpha`' one as both are incompatible. Default is no warping.

`-i, --freq-min=f`
: Specify band limiting and set the lower frequency bound to `f` Hz. Default is no band limiting.

`-u, --freq-max=f`
: Specify band limiting and set the upper frequency bound to `f` Hz. Default is no band limiting.

`-b, --fft-length=n`
: Set FFT length to `n` samples. The FFT length must be a power of two and greater than or equal to the number of samples in a frame. If FFT length is greater, the windowed frame samples are padded with zeroes before running the Fourier transform.

The following options are also available for `sfbcep`.

`-p, --num-ceps=n`
: Set the number of output cepstral coefficients to `n`. `n` must be less or equal to the number of channels in the filter bank. Default is 12.

`-r, --lifter=n`
: Set liftering parameter L to `n`. Default is no liftering.

`-e, --energy`
: Add log-energy to the feature vector.

`-s, --scale-energy=f`
: Scale energy according to $e_t = 1 + f(e_t - max_t(e_t))$. The way the maximum energy value is computed depends on whether '`--segment-length`' is specified or not.

`sfbank` supports the '`--delta`' and '`--acceleration`' options. In addition, `sfbcep` also supports the '`--cms`' and '`--normalize`' options. See Section 3.2 [Common options], page 13, for a description of these options and for additional ones.

3.4.2 LPC analysis tools

SPro provides two different tools, `slpc` and `slpcep`, for linear predictive analysis of speech signals.

Linear prediction coefficients

The tool `slpc` takes as input a waveform and output linear prediction derived features. For each frame, the signal is windowed after pre-emphasis and the generalized correlation is computed and further used to estimate the reflection and the prediction coefficients which can, in turn, be transformed into log area ratios or line spectrum frequencies. See Section 4.7.1 [Linear prediction], page 37, for mathematical details. The default is to output the linear prediction coefficients however reflection coefficients can be obtained with the '`--parcor`' option, log-area ratios with '`--lar`' option and line spectrum pairs with the '`--lsp`' one.

Optionally, the log-energy can be added to the feature vector. In `slpc`, the log-energy is taken as the linear prediction filter gain, which is also the variance of prediction error, and thresholded to be positive or null. The log-energies may be scaled to avoid differences between recordings using the '`--scale-energy`' option.

Linear prediction cepstrum

Program `slpcep` takes as input a waveform and outputs cepstral coefficients derived from the linear prediction filter coefficients. The linear prediction processing steps are as in `slpc` (see previous section) and cepstral coefficients are computed from the linear prediction coefficients using the recursion previously described. The required number of cepstral coefficients must be less then or equal to the prediction order.

As for `slpc`, the log-energy, taken as the gain of the linear prediction filter, can be added to the feature vectors.

Mean and variance normalization of the static cepstral coefficients can be specified with the global '`--cms`' and '`--normalize`' options but do not apply to log-energies. The normalizations can be global (default) or based on a sliding window whose length is specified with '`--segment-length`'.

Finally, first and second order derivatives of the cepstral coefficients and of the log-energies can be appended to the feature vectors. When using delta features, the absolute log-energy can be suppressed using the '`--no-static-energy`' option.

PLP cepstrum

Program `splp` takes as input a waveform and outputs cepstral coefficients derived from a perceptual linear prediction analysis. Note that, although not explicitely mentioned in the program name, `splp` does output cepstral coefficients, not linear prediction coefficients. The LPC order must be less than or equal to the number of filters in the filter-bank while the number of cepstral coefficients must be less than or equal to the prediction order.

The log-energy is taken from the frame waveform as in the filter-bank tools.

Options

The following options are available for `slpc`, `slpcep` and `splp`.

`-n, --order=n`
: Specify the linear prediction analysis order. Default is 12.

`-a, --alpha=f`
: Use bilinear frequency warping and set the warping parameter a to `f` (`f` must be between 0 and 1). Default is no warping.

`-e, --energy`
: Add log-energy to the feature vector.

`-s, --scale-energy=f`
: Scale energy according to $e_t = 1 + f(e_t - max_t(e_t))$. The way the maximum energy value is computed depends on whether '`--segment-length`' is specified or not.

The following options are specific to `slpc`.

`-r, --parcor`
: Output reflection coefficients rather than linear prediction coefficients.

`-g, --lar` Output log area ratios rather than linear prediction coefficients.

`-p, --lsp` Output line spectrum pairs rather than linear prediction coefficients.

The following options are also available for `slpcep` and `splp`.

`-p, --num-ceps=n`
: Set the number of output cepstral coefficients to `n`. `n` must be less or equal to the number of channels in the filter bank. Default is 12.

`-r, --lifter=n`
: Set liftering parameter L to `n`. Default is no liftering.

`splp` supports all of the options of `sfbank` for the control of the filter-bank (number of filters, bandwidth, Mel frequency warping, etc.). The power spectrum compression factor can be specified using '`--compress`'.

Also, `slpcep` and `splp` support the '`--cms`' and '`--normalize`' normalization options as well as '`--delta`' and '`--acceleration`'. See Section 3.2 [Common options], page 13, for a description of these options and for additional ones.

3.5 Manipulating feature streams

SPro provides a tool, `scopy` for manipulating feature streams. More than a mere copy tool, `scopy` also allows to normalize features, add dynamic features, scale the features, apply a linear transformation to the feature vectors and extract some components of the feature vector. All of these operations are detailed below. In addition, `scopy` can import feature files from previous SPro release, export files to alien formats such as HTK, or view the content of an SPro feature file in text format.

3.5.1 Operations on feature streams

As mentioned in the introduction, `scopy` may be used for

a. mean and variance normalization,
b. dynamic features computation,
c. multiplicative scaling,
d. linear transformation, and
e. components extraction.

The two first transformations, i.e. normalization and dynamic feature computation, are actually done at once when loading the input features. If normalization is specified, the static coefficients, not including energy, are normalized before delta and acceleration features are computed. If dynamic feature are used, the static log-energy can be discarded using '`--no-static-energy`'. As in all the feature extraction tools, normalization is either global or based on a sliding window, depending on whether '`--segment-length`' was specified or not.

Multiplicative scaling is a simple operation which consists in multiplying every component of every feature vector by a scaling factor. This is sometimes used to reduce the variance of features with a high dynamic range in order to avoid numerical problems when computing a linear transformation for those features or when doing some modeling.

A linear transformation matrix can be specified using '`--transform`' to project the input feature vectors according to $y'(t) = Az(t)$, where $y'(t)$ is the transformed vector for frame t and $z(t)$ is a column vector containing the input feature frame $y(t)$ plus possibly some context frames[5]. For example, assuming a context size k, $z(t)$ will be the concatenation of input feature vectors $y(t-k)$ to $y(t+k)$. If m is the input feature dimension, possibly after adding the dynamic features if this was asked, and n the output dimension, the transformation matrix will have `nrows`$=n$ rows and `ncols`$=(2k+1)*m$ columns. The matrix A is stored in a text file with the following syntax

```
nrows ncols nsplice
A[1][0]      A[1][1]    .........   A[1][ncols]
                         .........
A[nrows][0]             .........   A[nrows][ncols]
```

where `nsplice` is the context size.

Component extraction consists in extracting some components of the feature vectors. The extraction pattern is specified using the '`--extract=str`' option where `str` is a comma

[5] Frames are duplicated at the (buffer) boundaries.

separated list of components to keep. The latter are specified either as a single component index or as a index range using a dash (‘-’). Component indices start at 1. For example, the command

```
scopy --extract=1-12,25-36 foo.prm bar.prm
```

could be used to extract components 1 to 12 and 25 to 36 from ‘foo.prm’ into ‘bar.prm’, which, one can imagine, would correspond to keeping the 12 static features and the 12 acceleration features, thus discarding the delta features.

When performing either linear transformation or component extraction, the content of the resulting feature vector can no longer be described using a feature description flag. Indeed, specifying if a vector as delta features after a linear transformation does make no sense. For this reason, the output stream description flag will be arbitrarily set to zero if at least one of this transformation is specified.

If several operations are specified, they are applied in the order in which they are listed above. Therefore, delta coefficients are computed before the linear transformation if both are specified. As for now, there is unfortunately no direct and easy way to change the order of these operations. In particular, it is not possible to add delta coefficients after linear transformation which is an operation that does not seem illogical. The easiest, though CPU consuming, way to change the processing order is to use `scopy` several times, possibly with pipes. For example, the line

```
scopy --transform=pca.mat foo.prm - | scopy -ZD - bar.prm
```

will apply the linear transformation stored in file ‘pca.mat’ to the feature vectors in ‘foo.prm’ (first `scopy`) and then remove the mean of the static features before adding the delta features and store the result in ‘bar.prm’ (second `scopy`).

3.5.2 Exporting features

Exporting feature streams to alien formats is also possible with `scopy`. Currently, three alien formats are supported, namely HTK[6], Sirocco[7] and ASCII text format.

Export to HTK and Sirocco file formats is only possible on seekable streams, i.e. regular files in which the C function `fseek` works. The reason for this constraint is that those formats include the number of frames in the header. Since the number of frames is not in the SPro header, `sopy` uses `fseek` to seek to the end of the input feature stream in order to determine the number of frames. As a consequence, it is not possible to export to one of these alien formats when reading from a pipe. On the other hand, no seek in the output file is therefore necessary and the output of `scopy` can be piped into another command. This is particularly usefull with HTK, where setting the environment variable `HPARMFILTER` to ‘`scopy -o HTK $ -`’, enables to read directly read SPro files with HTK. See section "*Input/Output via Pipes and Networks*" in the HTK 3.2 book for details.

Export to ASCII is useful to list in a (almost) human-readable way the content of a feature stream. In particular, combining the ASCII output with the ‘--info’ option which gives information about the content of the input stream before possible transformation. For example, the command

[6] HTK is a popular Hidden Markov Model Toolkit from Cambridge University, http://htk.eng.cam.ac.uk.
[7] Sirocco is a free large vocabulary speech recognition search engine, http://gforge.inria.fr/projects/sirocco

```
scopy -zi -ZDA foo.prm -
```

will produce the following output

```
frame_rate = 100.00
input_dimension = 12
input_qualifiers = <nil>
output_dimension = 36
output_qualifiers = ZDA
```

In the above example, the input file dimension is 12 is then modified to 36 by adding the dynamic coefficients ('`-ZDA`'). Note that possible transformations (e.g. linear transform, bin extraction) are *not* taken into account in the output dimension and qualifiers. For instance, '`scopy -zi -ZDA -x 1-3,7 foo.prm -`' will still come up with the same output as above.

As mentioned in Section 3.1 [File formats], page 11, SPro feature files are always in little endian byte order. On the contrary, exported files are written in the machine's natural byte order. As both HTK and Sirocco expects files in big-endian byte order[8], the option '`--swap`' can be used to swap the byte order before writing the file in alien file formats. This option is ignored if the output file format is ASCII (obviously) or SPro.

3.5.3 Importing from a previous SPro release

The option '`--compatibility`' is provided for compatibility and enables to read feature files from previous versions of SPro (SPro 3.* and before). When this option is used, the entire feature file is loaded into memory at once as this used to be the case in previous versions. Using this options with large files may therefore be quite memory consuming (and slow by the same occasion). All the processing capabilities (normalization, dynamic features, linear transform, etc.) remains possible when importing files from previous SPro versions.

3.5.4 Copy options

The following options are available in `scopy`:

`-c, --compatibility`
: Turn on compatibility and set the input file format to former SPro format. Default is SPro 5.0 format.

`-I, --bufsize=n`
: Set the I/O buffer size in kbytes. Default is 10 Mbytes. If '`--compatibility`' is specified, the specified buffer size applies only to the output buffer, the entire input data being loaded into memory.

`-i, --info`
: Print stream information.

`-z, --suppress`
: Suppress data output. If this option is turned on, no output is created. This option is provided mainly for use with '`--info`' in order to print the stream description flag or for diagnosis purposes.

[8] In HTK, this actually depends whether or not `NATURALREADORDER=T` was specified in your configuration file.

`-B, --swap`
: Swap byte order before writing new file. Byte swapping is only possible if the output format is either HTK or Sirocco (see '`--output-format`' below). Default is to use the machine's natural byte-order.

`-o, --output-format=str`
: Set the output format, where `str` is one of `ascii`, `htk` or `sirocco`. Default is the native SPro format.

`-m, --scale=f`
: Scale features, multiplying them by the scaling factor `f`.

`-t, --transform=str`
: Apply the linear transformation whose matrix is specified in file `str`.

`-x, --extract=str`
: Extract the specified components of the feature vector. The argument `str` is a comma separated list of components to extract, where the components are specified either as a single index or a range of indices specified using a dash ('`-`'). The index of the first component is 1.

`-s, --start=n`
: Start copying frames at frame index `n`. Frame numbers start with zero. Default is 0.

`-e, --end=n`
: End copying at frame index `n` (included). Frame numbers start with zero. Default is to copy to the end of stream.

4 The SPro library

This chapter describes the main functions of the SPro library and should be sufficient for most implementations using the library. For more details, the reader is invited to read the source code which is, and will probably ever be, the most detailed and up-to-date description of what a function does. In particular, the library header '`spro.h`' gives a lot of details about functions arguments. The SPro tools[1] are good example on the use of the library functions.

Basic type definitions are voluntarily *not* given in the manual. Wherever necessary, *accessors* are given to access the most crucial members of structured types and, unless not possible otherwise, direct access should be avoided as much as possible in order to ensure a better compatibility with future versions of the library. For sake of rapidity, these accessors are mostly macros rather than functions. These accessors are described in the relevant sections.

4.1 Waveform streams

This section describes functions related to waveforms, or equivalently signals. From now on, the term signal will be used as a synonym to waveform unless otherwise specified. Functions related to signals are usually prefixed with `sig_` and located in '`sig.c`' and '`misc.c`'.

4.1.1 Memory allocation

Waveforms, or signals, are stored in a variable whose type is `spsig_t`. This type is not intended for storing waveform *streams*, i.e. the entire waveform for a document, but rather the frame samples. Therefore, no I/O functions are provided for this data type. Every signal processing function which operates on a frame takes as input a variable of the type `spsig_t`. Memory allocation for a signal is performed using `sig_alloc` and released using `sig_free`.

`spsig_t *` **sig_alloc** (`unsigned long *`*n*) Function

Allocate memory for a signal containing *n* samples. Return a pointer to the allocated structure or `NULL` in case of error.

`void` **sig_free** (`spsig_t *`*p*) Function

Free memory allocated for a signal using `sig_alloc`.

4.1.2 Opening streams

Signals are usually read from a stream, i.e. a collection of samples, from which the frames are made. As the SPro library has been designed to process signals into feature vectors, signal streams are solely input streams and no output function is provided. Therefore, a signal stream is always opened in read mode. The following two functions are used to open a stream for reading and to close the stream when all is done. Reading frames from a stream is explained in the next section.

[1] Maybe to the exception of `scopy` which is a total mess.

`sigstream_t *` **sig_stream_open** (`const char` **fn*, `int` *fmt*, `float` *Fs*, `size_t` *nbytes*, `int` *swap*) — Function

Open stream in file *fn* in read mode, where the file format is *fmt*. If *fn* is `NULL`, input will be made from `stdin`. Valid file formats are `SPRO_SIG_PCM16_FORMAT`, `SPRO_SIG_ALAW_FORMAT`, `SPRO_SIG_ULAW_FORMAT`, `SPRO_SIG_WAVE_FORMAT` and `SPRO_SIG_SPHERE_FORMAT` if the library has been compiled to support the SPHERE file format. If *fmt* is `SPRO_SIG_PCM16_FORMAT`, `SPRO_SIG_ALAW_FORMAT` or `SPRO_SIG_ULAW_FORMAT`, the sample rate *Fs* (in Hz) must be specified. Otherwise, the sample rate is read from the header and *Fs* is ignored. The input buffer size is specified by *nbytes*, which means *nbytes* bytes will be allocated for input. If *swap* is non null, byte swapping is performed on the samples after reading them. Return a pointer to the opened signal stream or `NULL` in case of error.

`void` **sig_stream_close** (`sigstream_t` **f*) — Function

Close a signal stream opened with `sig_stream_open`, releasing allocated memory.

4.1.3 Reading frames

Though possible, accessing directly samples in the stream is not the purpose of signal streams in SPro. Indeed, speech processing is based on the processing of successive overlapping frames. The library provides function to access directly to frames, such as `get_next_sig_frame` which returns frame samples which can be weighted using `sig_weight`. Weighting vectors for standard signal processing windows are created using `set_sig_win`.

`int` **get_next_sig_frame** (`sigstream_t` **f*, `int` *ch*, `int` *l*, `int` *d*, `float` *k*, `sample_t` **buf*) — Function

Read next frame from channel *ch* in stream *f*. Frames are *l* samples long with a shift of *d* samples between successive frames. Frame samples are returned in the buffer *buf* which must have been previously allocated to contain at least *d* samples. The content of *buf* must be kept untouched between two successive calls since some of the samples reused due to the overlap. Argument *k* sets the pre-emphasis factor. Return 1 in case of success and 0 otherwise.

`float *` **set_sig_win** (`unsigned long` *N*, `int` *type*) — Function

Allocate and initialize a weighting vector of length *N* for the specified window type, where *type* is one of `SPRO_HAMMING_WINDOW`, `SPRO_HANNING_WINDOW` and `SPRO_BLACKMAN_WINDOW`. The window type `SPRO_NULL_WINDOW` is defined for the purpose of argument processing but is not a valid argument for this function. Return a pointer to the allocated vector or `NULL` in case of error.

`spsig_t *` **sig_weight** (`spsig_t` **s*, `sample_t` **buf*, `float` **w*) — Function

Weight the samples in *buf* according to the weights in *w*. The result is returned in the previously allocated signal *s* whose size must correspond to the buffer's length. Return a pointer *s*.

The following is a typical piece of code used to open a signal stream and loop on all the input frames of *N* samples every *D* samples[2].

[2] For increased readability, error checking has been removed from the allocations.

```
spfstream_t *f = sig_stream_open("foo.wav",
                                 SPRO_SIG_WAVE_FORMAT, 0, 10000, 0);
spsig_t *frame = sig_alloc(N);
float *w = set_sig_win(N, SPRO_HAMMING_WINDOW);
sample_t *buf = (sample_t *)malloc(N * sizeof(sample_t));

while (get_next_sig_frame(f, 1, N, D, 0.95, buf)) {
  sig_weight(frame, buf, w); /* weight signal */

  /* ... */

}

sig_stream_close(f);
sig_free(frame);
free(w);
free(buf);
```

4.1.4 Computing frame energy

Assuming the frame signal is centered, `sig_normalize` compute the frame energy and may perform energy normalization to unity.

`double` **sig_normalize** (`spsig_t` **s*, `int` *norm*) Function

Return the square root of the sum of the squared samples in *s*. If *norm* is not null, normalize the signal variance to unity.

4.2 Feature description flags

Feature description flags are used to describe the content of a feature vectors indicating information about mean and variance normalization, delta features, etc. See Section 4.3 [Feature streams], page 26, for details. In the library, such flags are represented as field of bits, coded as `long` integers. To avoid incomprehensible code, symbolic constants are defined for each piece of information possibly encoded in the feature description flag. Bit mask constants are of the form `WITHX`, where `X` is one of the letter `E`, `Z`, `R`, `D`, `A` or `N`. The constant `SPRO_EMPTY_FLAG`, equals to 0, can also be used to denote an empty flag.

The two functions `set_flag_bits` and `get_flag_bits` can be used to raise or check the presence of elements (bits) in the flags. Alternatively, logical operators can be used directly on the flag value. For example, the instruction

```
flag = flag | WITHZ;
```

will raise the bit corresponding to mean subtraction while `flag & WITHZ` will be true if the bit corresponding to `Z` is raised and false otherwise. However, we recommend using the two macros for compatibility purposes. Another way o set flags is via the function `sp_str_to_flag` which converts a string of characters to a flag. The dual operation is implemented in `sp_flag_to_str`.

`long` **set_flag_bits** (`long` *flag*, `long` *mask*) Macro

Set to one the bits specified by *mask* in the the feature description flag *flag*. Return the resulting stream description flag. For example, the following line

```
flag = set_flag_bits(flag, WITHZ | WITHR)
```

will raise the bits `WITHZ` and `WITHR` in *flag*, corresponding to mean and variance normalization respectively. Bits already raised in *flag* will be left untouched.

`long` **get_flag_bits** (`long` *flag*, `long` *mask*) Macro

Return a flag containing the bits which are raised both in *flag* and in *mask*. The macro can be used as a boolean expression. However, this can be tricky, particularly if *mask* is a logical expression by itself. In this case, `get_flag_bits` will be true if at least two corresponding bits are raised in *flag* and *mask*. For example, if *mask* has the value `(WITHZ | WITHR)`, `get_flag_bits` will return true if *flag* has either the `WITHZ` or `WITHR` bit raised, or, obviously, both. To check that both bits are raised, use the following test

```
if (get_flag_bits(flag, WITHZ | WITHR) == (WITHZ | WITHR)) {
  /* ... */
}
```

`long` **sp_str_to_flag** (`const char` **str*) Function

Convert *str* into a feature description flag, where *str* is a string of description letters among `E`, `Z`, `R`, `D`, `A` or `N`. Return a flag where the bits corresponding to the letters in *str* are raised.

`char *` **sp_flag_to_str** (`long` *flag*, `char` *str*[7]) Function

Convert *flag* into a string containing the corresponding feature description letters. This function is mainly for tracing. Return a pointer to *str*.

4.3 Feature streams

This section describes the functions related to input and output of feature vectors. The functions are divided into three categories, namely opening a feature stream, reading and writing features from or to a stream and seeking to a particular position in the stream. Feature stream functions are usually prefixed by `spf_stream_` and are located in '`spf.c`', '`misc.c`' and '`header.c`'.

4.3.1 Opening feature streams

This section describes in detail feature streams open and close mechanism. The section also explains how to access stream attributes, such as fields in the variable length header or the frame rate for streams in read mode.

Conversion flags

In SPro, conversions such as adding dynamic features, normalization or energy scaling are associated with streams since these are typically global operations which cannot be carried out at the frame level. Such conversions are indicated by a *conversion flag* which specifies how the input data should be converted before output. In read mode, input refers to the file content and output is what is returned from the read function while, in write mode, input refers to the input of the write function and output to the file content. The conversion flag is a flag which indicates the processing that must be done between the input and the output. The conversion flag is actually a feature description flag containing the bits that should be raised in the output feature description flag in addition to those already present in the input description flag. For example, if the conversion flag takes the value `(WITHZ|WITHA)` and the input feature description flag, e.g. as specified in the header of an input file, is `(WITHZ|WITHD)`, the resulting feature description for the input stream will be `(WITHZ|WITHD|WITHA)`.

Though not coded as a flag, conversion in feature streams may include energy scaling. As this is not coded in the stream header, one must be careful not to specify scaling twice. Energy scaling conversion is turned on using `set_stream_energy_scale`. In a very similar way, the function `set_stream_seg_length` can be used to specify segmental normalization or scaling. Both functions should be called between the call to open and the first call to read or write, depending on the stream mode, in order to be effective.

`float` **set_stream_energy_scale** (`spfstream_t *`*f*, `float` *s*) Macro

Turn on energy scaling for stream *f* with a scale factor *s*. A null value of *s* disable energy scaling. This is the default value when the stream is opened. The function must be called after opening the stream and before any I/O operation on the stream. Return *s*.

`long` **set_stream_seg_length** (`spfstream_t *`*f*, `long` *length*) Macro

Turn on segmental normalization and scaling for stream *s* with a segment length of *length* frames. A null value of *length* disable energy scaling. This is the default value when the stream is opened. The function must be called after opening the stream and before any I/O operation on the stream. Return *length*.

Opening for I/O

As opposed to signal streams, feature streams can be either in read or write mode. Since the arguments are quite different in both cases, two different functions are provided, namely `spf_input_stream_open` and `spf_input_stream_open`. The function `spf_stream_close` is common to input and output streams.

Feature streams have very important attributes, such as the dimension, the feature description flag, the frame rate or the variable header, for which accessors are provided. Macros to access the most important attributes are documented here under.

`spfstream_t *` **spf_input_stream_open** (`const char *`*name*, `long` *flag*, `size_t` *nbytes*) Function

Open a feature stream associated to file *name* for reading with an associated buffer of *nbytes* bytes. Features read from *name* are converted according *flag*. See above for details on convertion flags. Return a pointer to the feature stream.

`spfstream_t *` **spf_output_stream_open** (`const char` **name*, `unsigned short` *dim*, `long` *iflag*, `long` *cflag*, `float` *Fs*, `const spfield_t *hd`, `size_t` *nbytes*) Function

Open a feature stream associated to file *name* for writing with a buffer of *nbytes* bytes. The input features, i.e. features added to the stream via `spf_stream_write`, dimension is *dim* with a corresponding feature description flag *iflag* and a frame rate of *Fs* Hz.. Conversion between the input features and the actual features written to file is specified by *cflag*. See above for details on conversion flags. Fields in the variable length header can be added via a possibly `NULL` array of fields *hd*, where *hd* is a NULL terminated array of `{char *name; char *value;}` elements. See example below. Return a pointer to the feature stream.

`void` **spf_output_stream_open** (`spfstream_t` **f*) Function

Close feature stream *f* opened with one of the `spf_*_stream_open` function, releasing allocated memory.

Accessing stream attributes

Stream attributes, such as dimension, fields in the variable length header, frame rate can be accessed using the following accessors.

`char *` **spf_stream_name** (`spfstream_t` **f*) Macro

Return a pointer to the filename associated with stream *f*. If the stream has no associated filename, i.e. I/O via `stdin` and `stdout`, return NULL.

`float` **spf_stream_rate** (`spfstream_t` **f*) Macro

Return the frame rate in Hz for stream *f*.

`unsigned short` **spf_stream_dim** (`spfstream_t` **f*) Macro

Return the feature vector dimension for stream *f*. The dimension corresponds to the dimension of the feature vectors possibly after conversion if the stream has a conversion flag set. For input streams, the dimension is therefore the dimension of the feature vectors returned by `get_next_spf_stream` while, for output stream, the dimension is the dimension as in the output header.

`long` **spf_stream_flag** (`spfstream_t` **f*) Macro

Return the feature description flag for stream *f*. The returned flag is taken after conversion, if any. For input streams, the flag describes the feature vectors returned by `get_next_spf_stream` while, for output stream, the flag is the output header's flag.

`spfheader_t *` **spf_stream_header** (`spfstream_t` **f*) Macro

Return a pointer to the (possibly empty) variable length header for stream *f*.

`char *` **spf_header_get** (`spfheader_t` **header*, `const char` **name*) Function

Return a pointer to the value of the attribute *name* in *header*. Return `NULL` if there are no attribute *name*.

`char *` **spf_header_get** (`spfheader_t *`*header*, `const char *`*name*) Function

Return a pointer to the value of the attribute *name* in *header*. Return `NULL` if there are no attribute *name*.

`int` **spf_header_add** (`spfheader_t *`*header*, `const spfield_t *`*tab*) Function

Add fields in *tab* to *header*, where *tab* is a `NULL` terminated array of `{char *name; char *value;}` elements. For example, the following code

```
spfheader_t *header = spf_header_init(NULL);
spfield_t tab[] = {
  {"snr", "20 dB"},
  {"date", "July 29, 2003"},
  {NULL , NULL}
};
spf_header_add (header, tab);
```

would create an empty header (undocumented function `spf_header_init`) and add the two fields '`snr`' and '`date`' to the header along with the corresponding values. No control is performed over duplicate field names. If several fields with the same name are added, the first one will always be returned by `spf_header_get` and the remaining one ignored. Return the number of fields added to the header.

4.3.2 Reading and writing feature vectors

The functions documented in this section are provided to read from or write to feature streams. Reading can be done in one of two ways. You can either read vector by vector using `get_next_spf_vec` or read in at once all the data in the feature buffer using `spf_stream_read`. Writing can only be done vector by vector using `spf_stream_write`, unless accessing directly the stream buffer. See Section 4.4 [Storing features without streams], page 30, for details on this highly not recommended operation. In write mode, the feature are actually written to the output file when the buffer is full or when the stream is closed. However, function `spf_stream_flush` can be used to force the output to file by flushing the buffer.

Note that the two functions `spf_stream_read` and `spf_stream_write` are actually not dual functions. The first one fills in the buffer with as much data as possible while the second one writes some feature vectors in the stream buffer.

`unsigned long` **spf_stream_read** (`spfstream_t *`*f*) Function

Fill in stream *f* buffer, reading until the buffer is full or the end of stream. Return the number of frames read.

`spf_t *` **get_next_spf_vec** (`spfstream_t *`*f*) Function

Return a pointer to the next feature vector in stream *f* or `NULL` at the end of stream. See Section 4.3.3 [Seeking into a stream], page 30, for details on how to get a particular vector in the stream.

`unsigned long` **spf_stream_write** (`spfstream_t` **f*, `spf_t` **buf*, `unsigned long` *n*) — Function

Write *n* feature vectors concatenated in *buf* to stream *f*. The feature vector dimension in *buf* is the dimension specified when the stream was opened. Return the number of frames written.

`unsigned long` **spf_stream_flush** (`spfstream_t` **f*) — Function

Flush the buffer of stream *f*, forcing the feature vectors to be actually written to the output file. Flushing has no effect on input streams. Return the number of frames written.

4.3.3 Seeking into a stream

The I/O functions described above are mainly intended for linear input and output, i.e. for reading or writing feature vectors in a sequential way. Though this is the most common case in speech processing, accessing a particular feature vector directly is also very useful. Functions to seek to a specified feature vector in a stream are provided. Feature vectors are indexed starting from 0. In read mode, seeking to a particular frame *n* using `spf_stream_seek` means that a pointer to frame *n* is returned by the next call to `get_next_spf_vec`. In write mode, the next call to `spf_stream_write` will start writing at frame *n*, thus overwriting frame *n* and possibly the following if those frames add already been set.

`int` **spf_stream_seek** (`spfstream_t` **f*, `long` *offset*, `int` *whence*) — Function

Seek *offset* frames according to *whence* in stream *f*. The *whence* argument is similar to the last argument of the C function `fseek` and specifies the reference point for *offset*. If *whence* is equal to `SEEK_SET` (0), *offset* is relative to the first frame. If *whence* is equal to `SEEK_CUR` (1), *offset* is relative to the current frame in the stream. Positioning relative to the end of the stream is not possible since the stream length is not known. The offset can be positive to seek forward in time or negative to seek backward. Seeking is only possible if the file associated with *f* is a seekable device, which is not the case of `stdout` or `stdin`. Return 0 if seek was correct or an error code (`SPRO_STREAM_SEEK_ERR`) otherwise.

`unsigned long` **spf_stream_tell** (`spfstream_t` **f*) — Macro

Return the current position, i.e. frame index, in *f*.

`int` **spf_stream_rewind** (`spfstream_t` **f*) — Macro

Seek to the beginning of the stream. This is equivalent to `spf_stream_seek(f, 0, SEEK_SET)`. Return 0 upon success.

4.4 Storing features without streams

In some programs, one may find useful to compute and keep in memory feature vectors inside a program without accessing the disk. This is for example the case if you want to embed feature extraction into your own program. Feature streams are of course not adapted to such operations which should rely on the use of *feature buffers* to store the feature vectors. Feature buffers are buffers containing a collection of feature vectors of the same dimension. Nearly no accessors are available for the buffer structure `spfbuf_t` whose attributes can be referenced directly. The structure definition is as follows:

```
typedef struct {
  unsigned short adim;          /* allocated vector dimension     */
  unsigned short dim;           /* actual vector dimension        */
  unsigned long n;              /* number of vectors              */
  unsigned long m;              /* maximum number of vectors      */
  spf_t *s;                     /* pointer to features            */
} spfbuf_t;
```

Note that the allocated dimension may not be the actual dimension of the features stored in the buffer. In particular, this is useful for feature conversions. See Section 4.5 [Feature conversion], page 33. The attribute `m` is the maximum number of vectors of dimension `adim` that can be stored in the buffer. Feature vectors are stored concatenated in the feature *array* `s`. Scanning the buffer vectors, using the `adim`, is illustrated in an example below.

4.4.1 Buffer allocation

Functions are provided to allocate a buffer of a given size in bytes, resize for a given number of feature vectors and free a buffer.

`spfbuf_t *` **spf_buf_alloc** (`unsigned short` *dim*, `size_t` *size*) Function

Allocate memory for a buffer of *size* bytes. The maximum dimension of the elements in the buffer is *dim*, the maximum number of vectors in the buffer being determined according to *dim* and *size*. If *size* is null, an empty buffer is allocated with the buffer array (`buf->s`) set to `NULL`. Return a pointer to the allocated buffer.

`spf_t *` **spf_buf_resize** (`spfbuf_t *`*buf*, `unsigned long` *n*) Function

Resize buffer *buf* to contain exactly *n* vectors. The buffer array is extended (resp. reduced) if *n* is more (resp. less) than the current buffer size. In both cases, the current content of the buffer is left unchanged. If the current buffer is empty (size is 0 and array is `NULL`), the buffer array is allocated. This function can therefore be used to allocate a buffer for a given number of vectors rather than for a given size in bytes as in `spf_buf_alloc`. The following code is an example for allocating a buffer of 1000 feature vectors of dimension 33 using `spf_buf_resize`.

```
spfbuf_t *buf = spf_buf_alloc(33, 0); /* alloc. empty buffer  */
spf_buf_resize(&buf, 1000);           /* resize for 1000 vectors */
```

Return the address of the first element of the buffer array. Note that the attribute `buf->s` may be changed in `spf_buf_resize`.

`void` **spf_buf_free** (`spfbuf_t *`*buf*) Function

Free memory allocated to *buf*.

4.4.2 Accessing buffer elements

The best way to reach a particular vector in a buffer is to grab a pointer to the vector using `get_spf_buf_vec`. In addition, the function `spf_buf_append` can be used to append feature vectors to a buffer, possibly extending the buffer size if necessary.

`spf_t *` **get_spf_buf_vec** (`spfbuf_t` **buf*, `unsigned long` *index*) Function

Return a pointer to vector *index* in *buf*. As opposed to positions in feature streams, the frame index *index* here is relative to the buffer, starting at 0. Return NULL if *index* is out of bound.

`spf_t *` **spf_buf_append** (`spfbuf_t` **buf*, `spf_t` **v*, `unsigned short` *dim*, `unsigned long` *nmore*) Function

Append feature vector *v* of dimension *dim* to buffer. If the buffer is full and *nmore* is not null, the buffer maximum size is extended by *nmore* vectors. Otherwise, if *nmore* is null, the buffer is left unchanged and `NULL` is returned. If the buffer is empty, the input vector dimension *dim* will be checked upon the buffer dimension. Else, *dim* will be used to initialize the buffer dimension. In any case, *dim* must be less than or equal to the maximum dimension (`buf->adim`) for which the buffer has been allocated. Return a pointer to the appended vector in the buffer or `NULL` in case of error.

Access to the buffer elements via `get_spf_buf_vec` implies a multiplication. Scanning all the vectors in the buffer may result faster using a pointer to the buffer array which is recursively incremented. The following example illustrates this method and print to `stdout` the feature vectors in text format.

```
unsigned long i;
unsigned short j;
spf_t *p;

p = buf->s;

for (i = 0; i < spf_buf_length(buf); i++) {

  /* print vector at index i */
  fprintf(stdout, "index %lu", i);
  for (j = 0; j < spf_buf_dim(buf); j++)
    fprintf(stdout, " %8.4f", *(p+j));
  fprintf(stdout, "\n");

  /* move to next vector */
  p += buf->adim;
}
```

Note that the pointer increment is the allocated dimension `adim`, not the actual dimension `dim`. This example also illustrates the use of the two accessors macros `spf_buf_length` and `spf_buf_dim` which return the actual number of elements in the buffer and the actual feature vector dimension respectively.

4.4.3 Buffer I/O

If you need the following functions to read or write the content of a buffer to disk, you should be wondering why you are not using feature streams for I/Os! Feature buffers are provided to store features in the memory not for I/Os, for which using the feature streams, dedicated to this purpose, should always be preferred. Still want to use buffer for I/Os?

Ok, but don't say you have not been warned! In case you insist on buffer I/Os, the two functions `spf_buf_read` and `spf_buf_write` are provided respectively to read the buffer content from disk or to write the buffer content to disk.

`unsigned long` **spf_buf_read** (`spfbuf_t` **buf*, `FILE` **f*) Function

Read data from file *f* into the buffer, until the buffer maximum sized is reached or until the end of file, whichever occurs first. The vector dimension is taken from the buffer actual dimension given by `buf->dim`. Return the number of vectors read into the buffer.

`unsigned long` **spf_buf_write** (`spfbuf_t` **buf*, `FILE` **f*) Function

Write the content of *buf* to file *f*. Return the number of vectors actually written to file.

4.4.4 Buffers and streams

In feature streams, I/O functions clearly make use of a feature buffer. Accessing directly the element of the stream buffer using the buffer functions described above is therefore possible. A pointer to the stream buffer can be obtained using `spf_stream_buf`.

`spfbuf_t *` **spf_stream_buf** (`spfstream_t` **f*) Macro

Return a pointer to the buffer of stream *f*.

Unless you are quite familiar with SPro programming, **direct access to stream buffers is strongly discouraged** since direct buffer I/Os may result in corrupted stream position information. The main consequence of corrupted stream position information is that `spf_stream_seek` and `spf_stream_tell` will not work properly. Rather than direct access to the stream buffer, the use of `spf_stream_seek` and `get_next_spf_frame` to access a particular vector should always be preferred.

4.5 Feature conversion

Feature conversion is the process of modifying the feature description flag, for example, by normalizing the feature mean and variance or by adding dynamic features. In other word, converting features consist on modifying the input features to match a specified target feature description. See Section 4.2 [Feature description flags], page 25.

Changing the feature type, e.g. converting feature bank features to cepstral coefficients, is not considered as a feature conversion and is outside the scope of the function described in this section. See Section 4.7 [LPC-based functions], page 37, for details about changing the the feature type between various LPC representation. See Section 4.6 [FFT-based functions], page 35, for details about changing the filter-bank representation..

Feature conversions are global operations in the sense that the conversion applies to a collection of feature vectors rather than to isolated feature vectors. Therefore, the conversion function, `spf_buf_convert`, operates on a feature buffer, modifying at once all the buffer vectors and returning a buffer (possibly the same — see below) containing the new features. The conversion itself is as follows

a. copy static features into the output buffer, possibly excluding energy if required.
b. normalize mean and variance of the static features in the output buffer (energy, if present, is not normalized) if required
c. compute delta features for the output buffer if required
d. compute acceleration features for the output buffer if required
e.

Since conversion principally aims at normalizing the features and adding dynamic features, the latter are always recomputed from the static features, even if the input feature vectors already contain dynamic features. This means that, for example, when converting features with a description flag value of `WITHE|WITHD` to `WITHE|WITHD|WITHN`, delta features will be recomputed, even though this is not strictly necessary[3]!

Conversion can operate under three different modes, namely duplicate, replace and update. In duplicate mode, `spf_buf_convert` allocates the output buffer and leaves the input buffer unchanged. This mode can be used to duplicate a buffer, hence the name. In replace mode, `spf_buf_convert` allocates the output buffer and releases memory allocated for the input buffer, thus replacing somehow the input buffer by the output one. Note that due to reallocation, the buffer address may have changed after the call to `spf_buf_convert`. In replace mode, calls to the conversion functions should therefore always look like

```
buf = spf_buf_convert(buf, SPRO_EMPTY_FLAG, WITHD, 0,
                                          SPRO_CONV_REPLACE);
```

for the caller function to take into account the new address for `buf`. Finally, in update mode, the output buffer is the same as the input one and conversion is done *in place*. For this, buffer maximum dimension must be at least equal to the maximum of the input and output dimensions. Otherwise update conversion is impossible and an error is returned. In any of the three mode, `spf_buf_convert` returns a pointer to the output buffer.

`spf_t *` **spf_buf_convert** (`spfbuf_t *`*buf*, `long` *iflag*, `long` *oflag*, `unsigned long` *wl*, `int` *mode*) Function

Convert feature vectors in *buf* from *iflag* description to *oflag*. The normalization window length *wl* specifies the length for segmental normalization. If null, global normalization is performed. Otherwise, use a sliding window of *wl* frames centered around the current frame. The mode is either `SPRO_CONV_DUPLICATE`, `SPRO_CONV_REPLACE`, `SPRO_CONV_UPDATE`. Return a pointer to the buffer containing the converted data.

In addition to `spf_buf_convert`, the function `spf_buf_normalize` can be used to normalize the mean and variance of the features in a buffer. Similarly, the (fragile) function `spf_delta_set` can be used to compute the derivatives of some features in a buffer. Both functions are *generic* functions which should be used solely for the purpose of non-standard operations. For example, normalizing the dynamic features or the energy variance is not possible with `spf_buf_convert` but is possible with `spf_buf_normalize`. Though not exactly a conversion function, `scale_energy` is a generic function used to scale the energy coefficients in a buffer.

[3] This will probably change in future versions where we should try to reuse as much as possible of the input features. Meanwhile, you will have to do with things the way they are...

`int` **spf_buf_normalize** (`spfbuf_t *`*buf*, `unsigned short` *s*, `unsigned short` *e*, `unsigned long` *wl*, `int` *vnorm*) Function

Normalize features *s* to *e* included in *buf*, where *s* and *e* are bins in the feature vectors and starts at 0. If *vnorm* is non null, variance normalization is performed in addition to mean subtraction. The normalization window length *wl* specifies the length for segmental normalization. If null, global normalization is performed. Otherwise, use a sliding window of *wl* frames centered around the current frame. Return 0 upon success or an error code otherwise.

`int` **spf_delta_set** (`spfbuf_t *`*ibuf*, `unsigned short` *in_k*, `unsigned short` *d*, `spfbuf_t *`*obuf*, `unsigned short` *out_k*) Function

Compute derivatives of features in the input buffer *ibuf*, from bin *in_k* for *d* bins, writing the result from bin *out_k* in the output buffer *obuf*. The output buffer can be the same as the input buffer and must have been properly allocated. This function is fragile as no mermory check is performed. It is therefore not exported and one should rather use `spf_buf_convert` directly. Should you require this function, you need to define `_convert_c_` before the inclusion of '`spro.h`'.

`int` **scale_energy** (`spfbuf_t *`*buf*, `unsigned short` *j*, `float` *s*, `unsigned long` *wl*) Function

Scale feature at bin *j* in *buf* by the factor *s*. This function is intended for log-energy scaling and scales with respect to the maximum value. If *wl* is non null, segmental scaling using a sliding window of *wl* frames is done. Return 0 upon success.

4.6 FFT-based functions

This section documents all the functions related to Fourier analysis of speech signals.

4.6.1 Fourier transform

SPro implements a fast Fourier transform (FFT) algorithm as described in *P. Duhamel and M. Vetterli, Improved Fourier and Hartley Transform Algorithms: Application to CycliC Convolution of Real Data, IEEE Trans. on ASSP, 35(6), June 1987*. For sake of rapidity, the implementation is based on a pre-computed FFT kernel which is initialized by `fft_init`. Initializing the FFT kernel for a given FFT size is necessary before the first invocation of `fft`. In particular, this implicates that the kernel should be reinitialized whenever the FFT size changes. Memory allocated to the kernel is released using `fft-reset`.

`int` **fft_init** (`unsigned long` *n*) Function

Initialize the FFT kernel for length *n*. If *n* is null, reset the kernel. Otherwise (re)allocate a kernel for the specified length: if the kernel had previously been allocated with a different size and not reset, it is reallocated. Return 0 upon success.

`int` **fft** (`spsig_t *`*s*, `float *`*mod*, `float *`*phi*) Function

Fourier transform of signal *s* using the current kernel. If the length of *s* is less than the kernel size, *s* is padded with zeros. On the contrary, if the length of *s* is more than the kernel size, *s* is truncated. Note that no warning occurs in this case. Return the modulus in *mod* and the phase in *phi*. Both *mod* and *phi* must have been allocated to contain at least $N/2$ elements, where N is the kernel size. Either one can be `NULL`, in which case no value is returned. Return 0 upon success.

`int` **fft_reset** () — Macro

Reset memory allocated to the FFT kernel. This is a macro to `fft_init(0)` which always returns 0.

4.6.2 Filter-bank

Filter-bank analysis is a two step process. The first step consists in defining the filter-bank geometry, either with `set_mel_idx` or `set_alpha_idx`. Both functions set the indices in the FFT magnitude vector of the filters' cutoff frequencies according to the specified frequency warping. The second step is the Fourier transform and the filter-bank integration embedded in function `log_filter_bank`. Using `log_filter_bank` requires that the FFT kernel has been initialized previously.

`unsigned short` **set_mel_idx** (`unsigned short *`*n*`, float` *fmin*`, float` *fmax*`, float` *Fs*) — Function

Set cutoff frequencies indices for *n* filters in the bandwidth *fmin* — *fmax*, according to MEL frequency warping. Lower and upper frequency bounds, *fmin* and *fmax* are normalized frequencies between 0 and 0.5. If *fmax* is lower than or equal to *fmin*, the upper bound will be considered to be the Nyquist frequency (1/2). The signal sample rate *Fs* is given in Hz. Return a vector of *n*+2 indices or `NULL` in case of error.

`unsigned short` **set_alpha_idx** (`unsigned short *`*n*`, float` *a*`, float` *fmin*`, float` *fmax*) — Function

Set cutoff frequencies indices for *n* filters in the bandwidth *fmin* — *fmax*, according to the bilinear frequency warping specified by *a*. If *a* is null, no frequency warping is used. Lower and upper frequency bounds, *fmin* and *fmax*, are normalized frequencies between 0 and 0.5. If *fmax* is lower than or equal to *fmin*, the upper bound will be considered to be the Nyquist frequency (1/2). Return a vector of *n*+2 indices or `NULL` in case of error.

`int` **filter_bank** (`spsig_t *`*s*`, unsigned short` *n*`, unsigned short *`*idx*`, int` *usepower*`, int` *uselog*`, spf_t *`*e*) — Function

Apply *n* channel triangular filter-bank to signal *s*. The indices in the FFT module vector of the channels cutoff frequencies are given in *idx*, which should have been initialized with one of the `set_*_idx` functions above. Depending on the two boolean flags *usepower* and *uselog*, the power or the magnitude spectrum or log-spectrum can be computed and returned in vector *e*, previously allocated to contain at least *n* elements. Return 0 upon success.

`int` **log_filter_bank** (`spsig_t *`*s*`, unsigned short` *n*`, unsigned short *`*idx*`, spf_t *`*e*) Macro

This function is a macro to `filter_bank` which returns the log of the magnitude spectrum. The use of `log_filter_bank` is deprecated and is solely provided for sake of compatibility with previous versions of SPro.

`double *` `set_loudness_curve` (`unsigned short` *n*`, unsigned short *`*idx*`, float` *Fs*) Function

Allocate memory and initialize a loudness equalization filter for *n* filters. Centre frequencies of the filters are determined from the indexes *idx*, obtained from one of the `set_*_idx` functions, and converted in Hertz assuming a signal sample rate of *Fs*.

4.6.3 Cosine transform

As for the Fourier transform, discrete cosine transform (DCT) is a kernel based transformation. A DCT kernel for a given size is initialized using `dct_init` while the transformation itself is carried out by `dct`. The macro `dct_reset` resets the kernel.

`int` **dct_init** (`unsigned short` *n*`, unsigned short` *m*) Function

Initialize the DCT kernel for a transformation from dimension *n* to *m*. If either *n* or *m* is null, reset the kernel. Otherwise (re)initialize a kernel for the specified transformation.length. Return 0 upon success.

`int` **dct** (`spf_t *`*x*`, spf_t *`*y*) Function

Apply transformation to *x*, storing the result in *y*. Assuming the kernel was initialized with lengths *n* and *m*, *x* should contain at least *n* elements and *y* must have been previously allocated to contain at least *m* elements. Return 0 upon success.

`int` **dct_reset** () Macro

Reset memory allocated to the FFT kernel. This is a macro to `dct_init(0, 0)` which always returns 0.

4.7 LPC-based functions

This section documents functions related to LPC analysis of speech signals. The first part documents how to solve the LPC equations while the second one deals with transforming the LPC or PARCOR representation into a different one.

4.7.1 Linear prediction

Linear prediction is a two step process in which the first step is to compute the generalized correlation sequence (`sig_correl`) before solving the normal equations with *lpc* to obtain the prediction and reflection coefficients.

`int` **sig_correl** (`spsig_t *`*s*`, float` *a*`, float *`*r*`, unsigned short` *p*) Function

Compute generalized correlation for *s* according to the warping specified by *a*. If *a* is null, the autocorrelation is used. Return a correlation sequence of length *p*+1 via the previously allocated vector *r*. Return 0 upon success.

`void` **lpc** (`float` **r*, `unsigned short` *p*, `spf_t` **a*, `spf_t` **k*, `float` **e*) Function

Compute *p* prediction and reflection coefficients given the correlation sequence $r(0)$ to $r(p)$. Return the prediction coefficients in *a*, the reflection coefficients in *k* and the LPC filter gain in *e*. Both *a* and *k* must have been previously allocated to contain at least *p* elements while *e* is a pointer to a `float` scalar.

4.7.2 LPC conversion

Linear prediction can be converted into line spectrum frequencies (`lpc_to_lsf`) and LP-derived cepstral coefficients (*lpc_to_cep*). Reflection coefficients are converted into log-area ratio using `refc_to_lar`.

`int` **lpc_to_lsf** (`spf_t` **a*, `unsigned short` *p*, `spf_t` **lsf*) Function

Convert *p* linear prediction coefficients *a* into line spectrum frequencies. *lsf* must have been previously allocated to contain at least *p* elements. Return 0 upon success.

`void` **lpc_to_cep** (`spf_t` **a*, `unsigned short` *p*, `unsigned short` *n*, `spf_t` **c*) Function

Convert *p* linear prediction coefficients *a* into *n* cepstral coefficients *c*. *c* must have been previously allocated to contain at least *n* elements.

`void` **refc_to_lar** (`spf_t` **k*, `unsigned short` *p*, `spf_t` **g*) Function

Convert *p* reflection coefficients *k* into *p* log area ratios *g*. *g* must have been previously allocated to contain at least *p* elements.

4.8 Miscellaneous functions

This section documents a bunch of very useful functions. The two functions `spf_indexes` and `spf_tot_dim` are dedicated to manipulating the content of a feature vector. A feature vector contains various elements characterized by the description flag. `spf_indexes` lets you find out where the indices of the various elements in a feature vector given the description flag while `spf_tot_dim` computes the feature vector total dimension from the dimension of the static coefficients and the description flag.

The function `set_lifter` is a utility functions that allocates memory for a lifter vector and initializes the vector according to the lifter parameter.

`void` **spf_indexes** (`unsigned short` *idx*[9], `unsigned short` *dim*, `long` *flag*) Function

Set in *idx* the indices of each element characterizing a feature vector of dimension *dim* with a description *flag*. *idx* is a nine element vector containing indices in the feature vector and organized as follow

```
<    static     ><E><      delta    ><dE>< delta delta  ><ddE>
|   |  ...  |   |   |   |  ...  |   |   |   |  ...  |   |   |
  ^           ^   ^   ^           ^   ^   ^           ^   ^
  |           |   |   |           |   |   |           |   |
idx[0]      idx[1]| idx[3]      idx[4]| idx[6]      idx[7]|
                  |                   |                   |
                idx[2]              idx[5]              idx[8]
```

For example, the index of the energy feature in the feature vector is *idx*[2] while the index of the first delta feature in the feature vector is given by *idx*[3]. With the exception of *idx*[0] which should always be equal to 0, an index value of 0 means that an element is not present in the feature vector. For example, a call to

```
spf_indexes(idx, 25, WITHE | WITHD | WITHN)
```

would return the following index vector

```
idx = { 0, 11, 0, 12, 23, 24, 0, 0, 0 }
```

Assuming *p* is a pointer to a feature vector, the 12 static features range from *p*[0] to *p*[11], no static log-energy is present (`WITHN`), delta features are from *p*[12] to *p*[23] and delta log-energy can be accessed at *p*[24].

`unsigned short` **spf_tot_dim** (`unsigned short` *sdim*, `long` *flag*) Function

Return the feature vector total dimension given the dimension of the static coefficients *sdim* (excluding energy) and the feature description *flag*.

`float *` **set_lifter** (`int` *l*, `unsigned short` *n*) Function

Return a pointer to a vector containing *n* coefficients for a lifter of parameter *l*.

5 Quick reference guide

This chapter is meant as a reference guide for all the SPro tools, summarizing the syntax, synopsis and options. This is actually a printed version of the online help message obtained with '`--help`'.

5.1 `sfbank`

Usage

```
sfbank [options] ifile ofile
```

Synopsis

Filter bank analysis of the input signal.

Options

`-F, --format=str`
: Specify the input waveform file format. Available formats are '`PCM16`', '`ALAW`', '`ULAW`', '`wave`' or '`sphere`'. Default: '`PCM16`'.

`-f, --sample-rate=f`
: Set input waveform sample rate to `f` Hz for '`PCM16`', '`ALAW`' or '`ULAW`' waveform files. Default: 8 kHz.

`-x, --channel=n`
: Set the channel to consider for feature extraction. Default: 1.

`-B, --swap`
: Swap the input waveform samples.

`-I, --input-bufsize=n`
: Set the input buffer size to `n` kbytes. Default: 10 Mb.

`-O, --output-bufsize=n`
: Set the output buffer size to `n` kbytes. Default: 10 Mb.

`-H, --header`
: Output variable length header.

`-k, --pre-emphasis=f`
: Set the pre-emphasis coefficient to `f`. Default: 0.95.

`-l --length=f`
: Set the analysis frame length to `f` ms. Default: 20.0 ms.

`-d, --shift=f`
: Set the interval between two consecutive frames to `f` ms. Default: 10.0 ms.

`-w, --window=str`
: Specify the waveform weighting window. Available windows are '`Hamming`', '`Hanning`', '`Blackman`' or '`none`'. Default: '`Hamming`'.

`-n, --num-filters=n`
: Set the number of channels in the filter bank. Default: 24.

`-a, --alpha=f`
: Set the bilinear frequency warping factor to `f`. Default: 0.

`-m, --mel`
: Use Mel frequency warping. Overwrites '`--alpha`'.

`-i, --freq-min=f`
: Set the lower frequency bound to `f` Hz. Default: 0 Hz.

`-u, --freq-max=f`
: Set the upper frequency bound to `f` Hz. Default: Niquist.

`-b, --fft-length=n`
: Set FFT length to `n` samples. Default: 512.

`-D, --delta`
: Add first order derivatives to the feature vector.

`-A, --acceleration`
: Add second order derivatives to the feature vector. Requires '`--delta`'.

`-v, --verbose`
: Turn on verbose mode

`-h, --help`
: Print a help message for the tool and exit.

`-V, --version`
: Print version information and exit.

5.2 sfbcep

Usage

```
sfbcep [options] ifile ofile
```

Synopsis

Filter-bank based cepstral analysis of the input signal.

Options

`-F, --format=str`
: Specify the input waveform file format. Available formats are 'PCM16', 'ALAW', 'ULAW', 'wave' or 'sphere'. Default: 'PCM16'.

`-f, --sample-rate=f`
: Set input waveform sample rate to `f` Hz for 'PCM16', 'ALAW' or 'ULAW' waveform files. Default: 8 kHz.

`-x, --channel=n`
: Set the channel to consider for feature extraction. Default: 1.

`-B, --swap`
: Swap the input waveform samples.

`-I, --input-bufsize=n`
: Set the input buffer size to `n` kbytes. Default: 10 Mb.

`-O, --output-bufsize=n`
: Set the output buffer size to `n` kbytes. Default: 10 Mb.

`-H, --header`
: Output variable length header.

`-k, --pre-emphasis=f`
: Set the pre-emphasis coefficient to `f`. Default: 0.95.

`-l --length=f`
: Set the analysis frame length to `f` ms. Default: 20.0 ms.

`-d, --shift=f`
: Set the interval between two consecutive frames to `f` ms. Default: 10.0 ms.

`-w, --window=str`
: Specify the waveform weighting window. Available windows are 'Hamming', 'Hanning', 'Blackman' or 'none'. Default: 'Hamming'.

`-n, --num-filters=n`
: Set the number of channels in the filter bank. Default: 24.

`-a, --alpha=f`
: Set the bilinear frequency warping factor to `f`. Default: 0.

`-m, --mel`
: Use Mel frequency warping. Overwrites '`--alpha`'.

`-i, --freq-min=f`
: Set the lower frequency bound to `f` Hz. Default: 0 Hz.

`-u, --freq-max=f`
: Set the upper frequency bound to `f` Hz. Default: Niquist.

`-b, --fft-length=n`
: Set FFT length to `n` samples. Default: 512.

`-p, --num-ceps=n`
: Set the number of output cepstral coefficients to `n`. `n` must be less or equal to the number of channels in the filter bank. Default: 12.

`-r, --lifter=n`
: Set liftering parameter to `n`. Default: 0.

`-e, --energy`
: Add log-energy to the feature vector.

`-s, --scale-energy=f`
: Set scale energy factor. The way the maximum energy value is computed depends on whether '`--segment-length`' is specified or not.

`-Z, --cms`
: Cepstral mean subtraction. Default: no.

`-R, --normalize`
: Variance normalization (requires '`--cms`'). Default: no.

`-L, --segment-length=n`
: Set normalization and energy scaling segment length.

`-D, --delta`
: Add first order derivatives to the feature vector.

`-A, --acceleration`
: Add second order derivatives to the feature vector. Requires '`--delta`'.

`-N, --no-static-energy`
: Remove static log-energy from feature vector (requires '`--delta`').

`-v, --verbose`
: Turn on verbose mode

`-h, --help`
: Print a help message for the tool and exit.

`-V, --version`
: Print version information and exit.

5.3 slpc

Usage

```
slpc [options] ifile ofile
```

Synopsis

Variable resolution AR modeling of the input signal.

Options

`-F, --format=str`
: Specify the input waveform file format. Available formats are '`PCM16`', '`ALAW`', '`ULAW`', '`wave`' or '`sphere`'. Default: '`PCM16`'.

`-f, --sample-rate=f`
: Set input waveform sample rate to `f` Hz for '`PCM16`', '`ALAW`' or '`ULAW`' waveform files. Default: 8 kHz.

`-x, --channel=n`
: Set the channel to consider for feature extraction. Default: 1.

`-B, --swap`
: Swap the input waveform samples.

`-I, --input-bufsize=n`
: Set the input buffer size to `n` kbytes. Default: 10 Mb.

`-O, --output-bufsize=n`
: Set the output buffer size to `n` kbytes. Default: 10 Mb.

`-H, --header`
: Output variable length header.

`-k, --pre-emphasis=f`
: Set the pre-emphasis coefficient to `f`. Default: 0.95.

`-l --length=f`
: Set the analysis frame length to `f` ms. Default: 20.0 ms.

`-d, --shift=f`
: Set the interval between two consecutive frames to `f` ms. Default: 10.0 ms.

`-w, --window=str`
: Specify the waveform weighting window. Available windows are '`Hamming`', '`Hanning`', '`Blackman`' or '`none`'. Default: '`Hamming`'.

`-n, --order=n`
: Set the prediction order. Default: 12.

`-a, --alpha=f`
: Set the bilinear frequency warping factor to `f`. Default: 0.

`-r, --parcor`
: Output reflection coefficients rather than linear prediction coefficients. Default: LPC.

`-g, --lar` Output log area ratios rather than linear prediction coefficients. Default: LPC.

`-p, --lsp` Output line spectrum pairs rather than linear prediction coefficients. Default: LPC.

`-e, --energy`
: Add log-energy to the feature vector, where the energy is the LPC filter gain.

`-s, --scale-energy=f`
: Set scale energy factor. The way the maximum energy value is computed depends on whether '`--segment-length`' is specified or not.

`-v, --verbose`
: Turn on verbose mode

`-h, --help`
: Print a help message for the tool and exit.

`-V, --version`
: Print version information and exit.

5.4 `slpcep`

Usage

`slpcep [options] ifile ofile`

Synopsis

Linear prediction based cepstral analysis of the input signal.

Options

`-F, --format=str`
: Specify the input waveform file format. Available formats are '`PCM16`', '`ALAW`', '`ULAW`', '`wave`' or '`sphere`'. Default: '`PCM16`'.

`-f, --sample-rate=f`
: Set input waveform sample rate to `f` Hz for '`PCM16`', '`ALAW`' or '`ULAW`' waveform files. Default: 8 kHz.

`-x, --channel=n`
: Set the channel to consider for feature extraction. Default: 1.

`-B, --swap`
: Swap the input waveform samples.

`-I, --input-bufsize=n`
: Set the input buffer size to `n` kbytes. Default: 10 Mb.

`-O, --output-bufsize=n`
: Set the output buffer size to `n` kbytes. Default: 10 Mb.

`-H, --header`
: Output variable length header.

`-k, --pre-emphasis=f`
: Set the pre-emphasis coefficient to `f`. Default: 0.95.

`-l --length=f`
: Set the analysis frame length to `f` ms. Default: 20.0 ms.

`-d, --shift=f`
: Set the interval between two consecutive frames to `f` ms. Default: 10.0 ms.

`-w, --window=str`
: Specify the waveform weighting window. Available windows are '`Hamming`', '`Hanning`', '`Blackman`' or '`none`'. Default: '`Hamming`'.

`-n, --order=n`
: Set the prediction order. Default: 12.

`-a, --alpha=f`
: Set the bilinear frequency warping factor to `f`. Default: 0.

`-p, --num-ceps=n`
: Set the number of output cepstral coefficients to `n`. `n` must be less or equal to the number of channels in the filter bank. Default: 12.

`-r, --lifter=n`
: Set liftering parameter to `n`. Default: 0.

`-e, --energy`
: Add log-energy to the feature vector.

`-s, --scale-energy=f`
: Set scale energy factor. The way the maximum energy value is computed depends on whether '`--segment-length`' is specified or not.

`-Z, --cms`
: Cepstral mean subtraction. Default: no.

`-R, --normalize`
: Variance normalization (requires '`--cms`'). Default: no.

`-L, --segment-length=n`
: Set normalization and energy scaling segment length.

`-D, --delta`
: Add first order derivatives to the feature vector.

`-A, --acceleration`
: Add second order derivatives to the feature vector. Requires '`--delta`'.

`-N, --no-static-energy`
: Remove static log-energy from feature vector (requires '`--delta`').

`-v, --verbose`
: Turn on verbose mode

`-h, --help`
: Print a help message for the tool and exit.

`-V, --version`
: Print version information and exit.

5.5 `splp`

Usage

```
splp [options] ifile ofile
```

Synopsis

Perceptual linear prediction based cepstral analysis of the input signal.

Options

`-F, --format=str`
: Specify the input waveform file format. Available formats are ‘`PCM16`’, ‘`ALAW`’, ‘`ULAW`’, ‘`wave`’ or ‘`sphere`’. Default: ‘`PCM16`’.

`-f, --sample-rate=f`
: Set input waveform sample rate to `f` Hz for ‘`PCM16`’, ‘`ALAW`’ or ‘`ULAW`’ waveform files. Default: 8 kHz.

`-x, --channel=n`
: Set the channel to consider for feature extraction. Default: 1.

`-B, --swap`
: Swap the input waveform samples.

`-I, --input-bufsize=n`
: Set the input buffer size to `n` kbytes. Default: 10 Mb.

`-O, --output-bufsize=n`
: Set the output buffer size to `n` kbytes. Default: 10 Mb.

`-H, --header`
: Output variable length header.

`-k, --pre-emphasis=f`
: Set the pre-emphasis coefficient to `f`. Default: 0.95.

`-l --length=f`
: Set the analysis frame length to `f` ms. Default: 20.0 ms.

`-d, --shift=f`
: Set the interval between two consecutive frames to `f` ms. Default: 10.0 ms.

`-w, --window=str`
: Specify the waveform weighting window. Available windows are ‘`Hamming`’, ‘`Hanning`’, ‘`Blackman`’ or ‘`none`’. Default: ‘`Hamming`’.

`-n, --num-filters=n`
: Set the number of channels in the filter bank. Default: 24.

`-a, --alpha=f`
: Set the bilinear frequency warping factor to `f`. Default: 0.

`-m, --mel`
: Use Mel frequency warping. Overwrites '`--alpha`'.

`-i, --freq-min=f`
: Set the lower frequency bound to `f` Hz. Default: 0 Hz.

`-u, --freq-max=f`
: Set the upper frequency bound to `f` Hz. Default: Niquist.

`-b, --fft-length=n`
: Set FFT length to `n` samples. Default: 512.

`-c, --compress=f`
: Set the power spectrum compression factor. Default: 3.

`-q, --order=n`
: Set the prediction order. Default: 12.

`-a, --alpha=f`
: Set the bilinear frequency warping factor to `f`. Default: 0.

`-p, --num-ceps=n`
: Set the number of output cepstral coefficients to `n`. `n` must be less or equal to the number of channels in the filter bank. Default: 12.

`-r, --lifter=n`
: Set liftering parameter to `n`. Default: 0.

`-e, --energy`
: Add log-energy to the feature vector.

`-s, --scale-energy=f`
: Set scale energy factor. The way the maximum energy value is computed depends on whether '`--segment-length`' is specified or not.

`-Z, --cms`
: Cepstral mean subtraction. Default: no.

`-R, --normalize`
: Variance normalization (requires '`--cms`'). Default: no.

`-L, --segment-length=n`
: Set normalization and energy scaling segment length.

`-D, --delta`
: Add first order derivatives to the feature vector.

`-A, --acceleration`
: Add second order derivatives to the feature vector. Requires '`--delta`'.

`-N, --no-static-energy`
: Remove static log-energy from feature vector (requires '`--delta`').

`-v, --verbose`
: Turn on verbose mode

`-h, --help`
: Print a help message for the tool and exit.

`-V, --version`
: Print version information and exit.

5.6 `scopy`

Usage

```
scopy [options] ifile ofile
```

Synopsis

Copy input file to output file making necessary conversions. Possible conversions are normalization, dynamic features, scaling, linear transformation and component extraction.

Options

`-c, --compatibility`
: Turn on compatibility and set the input file format to former SPro format. Default is SPro 5.0 format.

`-I, --bufsize=n`
: Set the I/O buffer size in kbytes. Default is 10 Mbytes. If '`--compatibility`' is specified, the specified buffer size applies only to the output buffer, the entire input data being loaded into memory.

`-i, --info`
: Print stream information.

`-z, --suppress`
: Suppress data output. If this option is turned on, no output is created. This option is provided mainly for use with '`--info`' in order to print the stream description flag or for diagnosis purposes.

`-B, --swap`
: Swap byte order before writing new file. Byte swapping is only possible if the output format is either HTK or Sirocco (see '`--output-format`' below). Default is to use the machine's natural byte-order.

`-o, --output-format=str`
: Set the output format, where `str` is one of `ascii`, `htk` or `sirocco`. Default is the native SPro format.

`-H, --header`
: Output variable length header.

`-R, --normalize`
: Variance normalization (requires '`--cms`'). Default: no.

`-L, --segment-length=n`
: Set normalization and energy scaling segment length.

`-D, --delta`
: Add first order derivatives to the feature vector.

`-A, --acceleration`
: Add second order derivatives to the feature vector. Requires '`--delta`'.

`-N, --no-static-energy`
: Remove static log-energy from feature vector (requires '`--delta`').

`-m, --scale=f`
: Scale features, multiplying them by the scaling factor `f`.

`-t, --transform=str`
: Apply the linear transformation whose matrix is specified in file `str`.

`-x, --extract=str`
: Extract the specified components of the feature vector. The argument `str` is a comma separated list of components to extract, where the components are specified either as a single index or a range of indices specified using a dash ('`-`'). The index of the first component is 1.

`-s, --start=n`
: Start copying frames at frame index `n`. Frame numbers start with zero. Default is 0.

`-e, --end=n`
: End copying at frame index `n` (included). Frame numbers start with zero. Default is to copy to the end of stream.

6 Changes

6.1 History

Here is a little bit of history before going into the details of the changes between the two last version of SPro.

I started the SPro project in 1996 while working at ELAN Informatique. At the time, SPro was nothing but a simple linear prediction analysis library for a CELP coder.

After I left this company to go as a Ph. D. Student at ENST Paris, the project rapidly turned into a speech processing toolkit to design front-end processing for speech and speaker recognition algorithms. In particular, all those nice variable resolution spectral analysis programs were developed for my work there. I left ENST with SPro version 3.2, a rather stable version of the toolkit but dependent upon non GPL code and upon other toolkits I used to maintain.

After a short pause, the project restarted when I joined IRISA as a CNRS fellow researcher. Version 3.2 then quickly turned into version 3.3 which is the first truly GPL stand-alone distribution of SPro. Version 3.3 also introduced the use of the `configure` script which has made SPro developer's life easier since.

Finally, it took quite a long time and several non fully documented (3.3.1) or non distributed (3.3.2) intermediate versions of SPro before the major rewrite that lead to version 4.0. Version 4.0 had basically the same functionalities as had versions 3.x with the immense advantage that the new version can handle signals of virtually infinite length via the use of signal and feature streams. Implementing streamed I/O mechanisms for waveforms and features required rewriting a lot of functions and changing the SPro feature file format, **thus loosing both the command and the library compatibility** (see note on the compatibility below). I took this opportunity of a major rewrite to undergo modifications I had been willing to do for a long time.

Finally, in release 4.1, I added PLP analysis and a few goodies (see changes below).

Apart from a few features that I would like to add to SPro, the toolkit is pretty stable and no major modifications are scehduled in the near (or even mid-term) future.

6.2 Changes from previous version

The distribution license has been changed from GPL to a more permissive MIT License. Release 5.0 is in all point similar to 4.1 except for the license change.

6.3 Compatibility

Release 5.0 is fully compatible with 4.* releases.

In case you did not get it, **from version 4.0, the compatibility is lost with previous releases**. As mentioned previously, the main reason for a new organization of SPro starting with version 4.0 is the ability to process arbitrary length streams.

To enable arbitrary length streams, the feature file format has changed and feature files (formerly known as data files) generated with a version of SPro prior to 4.0 can not be used directly in 4.0 and above versions. However, the `scopy` tool provides a compatibility option which enables the import of feature files from previous SPro releases.

The lack of compatibility also extends to the library. Programs based on former releases of the library will not compile anymore.

Index