Pm
A Library for Additive Analysis/Transformation/Synthesis

Guillermo García[1]

Version 1.2

Library Version History[2]

Version	Date			Author	Changes
0.9	2.	July	1994	Guillermo García	Original version
1.0	21.	May	1997	Rolf Woehrmann	Reworked for Mac/Unix
1.1	14.	May	1998	Diemo Schwarz	Added PmGetAllPartial(Freq|Ampl|Phase)
1.2.1	22.	July	1999	Diemo Schwarz	Added SDIF I/O, initialisation, file handling
1.2.2	23.	September	1999	Diemo Schwarz	Added bpf I/O, changed PmAddSynt prototype

[1]Documentation formatted and converted to HTML by Diemo Schwarz (`schwarz@ircam.fr`)

[2]This document generated September 27, 1999, Source control revision: $RCSfile: Pm.tex,v$, $Revision: 1.6$, $Date: 1999/09/27 14:07:55$, $Author: schwarz$

Contents

Alphabetical Index

1 Introduction

Pm is a C library for additive analysis, transformation, and synthesis of sound. Other signal models will be incorporated in the future.

This document describes the *Pm* library from the user's point of view. For the programmer, an automatically generated HTML documentation[1] of all data types and functions (even the internal ones) is available.

2 Data Types

Pm defines some specific data structures. Access to fields of these structures should always be made using the data interface functions provided by *Pm*.

The following data types are specific to *Pm*:

2.1 PmPartialSet

A set of partials defined at one given time.

Accessible parameters of a PartialSet are:

- time
- number of current partials
- index, frequency, amplitude and phase of each partial.
- allocation size: the maximum index value of a partial in the set.

2.2 PmFrameSet

A set of data frames.

A frame is defined as a collection of parameter sets defined at one given time.

A frame can hold parameter sets of different types, including PartialSet and C basic data types.

When a FrameSet is first created, it has a given number of allocated, empty frames. Frames can be filled with data strictly in sequential order, so as to avoid gaps of data in the middle of a FrameSet.

Filled frames will be called "active" frames from now on.

Times in a FrameSet must always be in increasing order. Otherwise, *Pm* functions are not warranted to work.

[1] `http://www.ircam.fr/equipes/analyse-synthese/pm`

A FrameSet is defined in a time interval [beginTime ; endTime]. This continuous-time limits may be different from the respective times of the first and last frame of the FrameSet. They are used in some editing functions (PmCutFrameSet[2], PmInsertFrameSet[3]) to keep the exact time distance between frames.

Accessible parameters of a FrameSet are:

GLOBAL LEVEL:

- begin time, end time.
- number of active frames.
- number of ParamSets.
- identifiers of created ParamSets.
- variable type, size of type, and name of a given ParamSet.

FRAME LEVEL:

- time of a given frame.
- allocated number of elements, active number of elements, and data pointer of a ParamSet in a given frame.
- data of a ParamSet, or of a group of ParamSets, in a given frame.

2.3 PmParamSetId

An identifier of a parameter set in a frame set.

2.4 PmBreakPoint

A simple breakpoint function using float data type.

---

3 Compilation

3.1 Using the Library

C files calling *Pm* functions must include the header file `Pm.h`:

```
#include "Pm.h"
```

You must link to the libraries `libPm.a` and the math library `libm.a` with:

```
-lPm -lm
```

[2]see section 4.10.1 page 31
[3]see section 4.10.2 page 32

To find them, the search paths for the compiler and the linker must include the *Pm* installation directories, at the moment, that's

```
-I/u/formes/include -L/u/formes/lib/$(ARCH)
```

3.2 Building the Library

In the sub-directory `src` of the *Pm* project directory, say `make` to build the library, the tools, and the parser. These can be built individually in their respective sub-directories `LibSrc`, `Tools`, and `Parser`.

3.3 Testing

The library, the tools, and the parser can be tested in directory `src/test`. Say `make` to run various test cases.

3.4 Installation of the Library

By saying `make install` the header file, the SDIF[4] type definitions, the library, and the programs are installed in `/u/formes/include`, `/u/formes/lib`, `/u/formes/lib/$(ARCH)`, and `/u/formes/bin/$(ARCH)`, respectively.

The installs can be initiated seperately with the targets `install-incl`, `install-lib` (includes the types file), and `install-prog`.

The installation root can be changed from `/u/formes` by setting the make-variable `PREFIX`:

```
make install PREFIX=/where/do/you/want/to/install/today
```

3.5 Documentation

The documentation is generated in the `doc` sub-directory. The `make` targets are: `tex`, `html`, and `cocoon`, for the printable user documentation, its HTML version, and the internal programmer documentation, respectively.

[4] `http://www.ircam.fr/sdif`

4 Functions: Synopsis and Description

4.1 Initialisation

4.1.1 PmInit

Synopsis
`int PmInit (void)`

Initialises the Pm library (especially the SDIF[5] part).

Returns 1 on success.

4.1.2 PmDeInit

Synopsis
`int PmInit (void)`

Deinitialises the Pm library (free static memory and deinit the SDIF[6] part).

Returns 1 on success.

4.2 File Handling

4.2.1 File Formats

Pm recognises the following file formats for I/O:

PM_ASCII Ascii format, as described in additive[7], usually with file extension .format

PM_BIN Binary format, as described in additive[8], usually with file extension .fmt

PM_SDIF SDIF[9] format, usually with file extension .sdif. With SDIF, there are several sub-formats to choose from, described at CNMAT's SDIF site[10].

PM_SDIF_TRACKS partial tracks (the default, when only PM_SDIF is used)

PM_SDIF_HARMONICS harmonic partials

PM_SDIF_PEAKS picked peaks

[5] http://www.ircam.fr/sdif
[6] http://www.ircam.fr/sdif
[7] http://www.ircam.fr/equipes/analyse-synthese/additive
[8] http://www.ircam.fr/equipes/analyse-synthese/additive
[9] http://www.ircam.fr/sdif
[10] http://cnmat.CNMAT.Berkeley.EDU/SDIF/FrameTypes.html

4.2.2 PmGetExtension

Synopsis

```
char *PmGetExtension (char *name)
```

Find the extension of a filename (the part after the last '.' of the filename part of the full path).

Returns Pointer to extension (after the '.'), or NULL if no extension found

---

4.2.3 PmNameToFormat

Synopsis

```
int PmNameToFormat (char *name)
```

Determine *Pm* format[11] from string name.

Returns format[12].

4.2.4 PmGetFileFormat

Synopsis

```
int PmGetFileFormat (char *name, int defnonsdif, int defsdif)
```

Determine *Pm* format[13] from extension of file given by name, unless it is an SDIF[14] file (determined by the first 4 characters in the file.

Returns format[15]`defsdif`, if file is an SDIF[16] file, otherwise, the format[17] determined by the extension in name, if none found `defnonsdif`.

4.2.5 PmOpenFile

Synopsis

```
FILE *PmOpenFile (char *name, int mode, int format[a])
```

[a] see section 4.2.1 page 11

Open file *name* with given mode (`PM_READ`, `PM_WRITE`, or `PM_APPEND`) and format[18].

Note that, on systems which have a popen() function, if the extension is (.gz), the file is automatically decompressed or compressed with *gzip*, using a pipe.

Returns opened file pointer or NULL on failure. Note that in case of an SDIF[19] format[20] the return value points to an Sdif file structure and must not be used except for Pm i/o.

---

[11] see section 4.2.1 page 11
[12] see section 4.2.1 page 11
[13] see section 4.2.1 page 11
[14] `http://www.ircam.fr/sdif`
[15] see section 4.2.1 page 11
[16] `http://www.ircam.fr/sdif`
[17] see section 4.2.1 page 11
[18] see section 4.2.1 page 11
[19] `http://www.ircam.fr/sdif`
[20] see section 4.2.1 page 11

4.2.6 PmCloseFile

Synopsis

```
void PmCloseFile (FILE *f, int format^a)
```

[a] see section 4.2.1 page 11

Close a file which was opened with format[21] `format`.

4.3 Breakpoints

4.3.1 PmCreateBreakPoint

Synopsis

```
PmBreakPoint PmCreateBreakPoint (int size)
```

Creates and returns a BreakPoint of size points.

Returns NULL if there is no available memory.

4.3.2 PmReallocBreakPoint

Synopsis

```
PmBreakPoint PmReallocBreakPoint (PmBreakPoint aBP, int size)
```

Reallocates input BreakPoint aBP up to size points.

Returns

If reallocation is successful, returns the same input BreakPoint aBP with reallocated number of points.

If reallocation fails, returns NULL and the size of aBP remains unchanged.

If aBP is NULL, returns a new BreakPoint of size points, or NULL if reallocation failed.

4.3.3 PmFreeBreakPoint

Synopsis

```
void PmFreeBreakPoint (PmBreakPoint aBP)
```

Frees aBP.

4.3.4 PmReadBreakPoint

Synopsis

```
int PmReadBreakPoint (FILE *Stream, PmBreakPoint aBP, int format_flag)
```

Reads aBP from the input Stream.

`format_flag` indicates the format of the input file:

PM_BIN or PM_BIN_BPF binary [x y] points

[21] see section 4.2.1 page 11

PM_ASCII or PM_ASCII_BPF ascii [x y] points on the same line . Comments can be placed after each point [x y] in a line.

PM_SDIF_BPF SDIF[22] format, by default reads 1FQ0 frames and matrices. If an SDIF[23] selection specification is appended to the file name, the stream, frame and matrix type therein will be read.

Returns [number of points] if a BreakPoint was read, -1 if there is no available memory to perform the operation.

4.3.5 PmWriteBreakPoint

Synopsis

```
void PmWriteBreakPoint (FILE *Stream, PmBreakPoint aBP, int format_flag)
```

Writes aBP to the output Stream.

`format_flag` indicates the format of the input stream (see above).

4.3.6 PmLoadBreakPoint

Synopsis

```
PmBreakPointPmLoadBreakPoint (char *name)
```

Creates and loads an entire bpf from a file given by name. The format is determined by checking if it is an SDIF[24] file, if not, `PM_ASCII_BPF` is assumed.

Returns the bpf read, which must be PmFreeBreakPoint'ed by the caller.

4.3.7 PmLoadBreakPointOrConst

Synopsis

```
PmBreakPointPmLoadBreakPointOrConst (char *nameorconst)
```

If nameorconst looks like a number, returns a constant PmBreakPoint with that value, else, uses it as filename and loads a bpf from it. This is especially useful for parsing command line arguments, which can then be transparently either a constant (e.g. maximum fundamental frequency to be considered) or a bpf from a file.

On load, the function tries to auto-detect the file format, if it is not SDIF[25], for now we assume `PM_ASCII_BPF`.

Returns: A freshly created PmBreakPoint (which must be PmFreeBreakPoint'ed by the calling program).

4.3.8 PmSaveBreakPoint

Synopsis

```
int PmSaveBreakPoint (char *name, PmBreakPointbp, int format)
```

[22] `http://www.ircam.fr/sdif`
[23] `http://www.ircam.fr/sdif`
[24] `http://www.ircam.fr/sdif`
[25] `http://www.ircam.fr/sdif`

Saves an entire bpf to file `name`, in format `format`. If the format is `PM_SDIF_BPF`, the SDIF[26] selection specification determines the SDIF[27] stream, frame and matrix type (1FQ0 by default). E.g. if `name = "foo.sdif::#4:1WRP/1BPF"` writes on stream 4 as 1BPF matrices in 1WRP frames.

Returns 1 on success.

4.3.9 PmEvalBreakPoint

Synopsis

```
float PmEvalBreakPoint (PmBreakPoint aBP, float x)
```

Evaluates the function aBP at X = x by linear interpolation and returns the corresponding Y value.

4.3.10 PmAddBreakPoint

Synopsis

```
int PmAddBreakPoint (PmBreakPoint aBP, float x, float y)
```

Adds a point [x ; y] to the function aBP.

Returns 1 if addition was successful, -1 if there is no available memory to add the point.

4.3.11 PmExtractBreakPoint

Synopsis

```
void PmExtractBreakPoint (PmBreakPoint aBP, float x)
```

Extracts (=deletes) from the function aBP the point whose x-coordinate is closest to x.

4.3.12 PmGetBreakPoint

Synopsis

```
int PmGetBreakPoint (PmBreakPoint aBP, int pos, float *px, float *py)
```

Writes to the addresses px and py the coordinates of the point in position=pos in the function aBP.

Returns -1 if pos is greater than the [nb_of_points - 1] of aBP, 1 otherwise.

4.3.13 PmGetBreakPointNb

Synopsis

```
int PmGetBreakPointNb (PmBreakPoint aBP)
```

Returns the number of points of the function aBP.

---

[26] `http://www.ircam.fr/sdif`
[27] `http://www.ircam.fr/sdif`

4.4 Partialsets

4.4.1 PmCreatePartialSet

Synopsis

`PmPartialSet PmCreatePartialSet (int max_index)`

Creates and returns a new PartialSet.

max_index is the greatest partial index the new PartialSet can initially hold. Reallocation of an existing PartialSet can be done later.

Returns NULL if there is no available memory.

4.4.2 PmReallocPartialSet

Synopsis

`PmPartialSetPmReallocPartialSet (PmPartialSet aPS, int new_max_index)`

Reallocates the input PartialSet aPS up to new_max_index.

Returns

If reallocation is successful, returns the same input PartialSet aPS with reallocated size.

If reallocation fails, returns NULL and the size of aPS remains unchanged.

If aPS is NULL, returns a new PartialSet of size new_max_index, or NULL if reallocation failed.

Most functions that operate on PartialSets provide for automatic reallocation. Only the PartialSet's data access functions require the application programmer to test allocation sizes before putting/getting data to/from a PartialSet.

4.4.3 PmFreePartialSet

Synopsis

`void PmFreePartialSet (PmPartialSet aPS)`

Frees PartialSet aPS.

4.4.4 PmReadPartialSet

Synopsis

`int PmReadPartialSet (FILE *Stream, PmPartialSet aPS, int format_flag)`

Reads one PartialSet aPS from the input Stream.

format_flag indicates the format[28]of the input stream.

Returns

1 if a PartialSet was read.

0 if EOF

-1 if there is no available memory to perform the operation.

[28]see section 4.2.1 page 11

4.4.5 PmWritePartialSet

Synopsis

```
int PmWritePartialSet (FILE *Stream, PmPartialSet aPS, int format_flag)
```

Writes one PartialSet aPS to the output Stream.

format_flag indicates the format[29]of the output stream.

Returns

1 if a PartialSet was written.

-1 if there is no available memory to perform the operation.

4.4.6 PmCopyPartialSet

Synopsis

```
void PmCopyPartialSet (PmPartialSet source, PmPartialSet destination)
```

Copies data from PartialSet source to PartialSet destination.

Returns 1 if data were successfully copied, -1 if there is no available memory to copy all source data.

4.4.7 PmDetectSpectralPeaks

Synopsis

```
int PmDetectSpectralPeaks (float *complex_fft, PmPartialSet peaks, float srate,
int fft_size, int window_size)
```

Fills peaks with peaks detected in the spectrum complex_fft.

Index values of PartialSet peaks do not correspond to partial indexes. They identify a peak in the spectrum.

Vector complex_fft contains FFT real part at even locations, and FFT imaginary part at odd locations.

srate : sampling rate

fft_size : number of complex fft points

window_size : number of points of the signal window

Returns

[number of peaks] if detection is successful.

-1 if there is no available memory to perform the operation.

4.4.8 PmSeeve

Synopsis

```
int PmSeeve (PmPartialSet peaks, PmPartialSet harms, float f0, float band, float
srate, int nbHarm)
```

[29]see section 4.2.1 page 11

Logical filtering of spectral peaks: selects the peak of greatest amplitude in each harmonic band.

peaks : input spectral peaks. Could be the output of PmDetectSpectralPeaks[30].

harms : output selected spectral peaks

f0 : fundamental frequency value

band : width of harmonic bands in pct (i.e. 100 means bandwidth = f0)

srate : sampling rate

nbHarm : max number of harmonics

Returns ?

4.4.9 PmPresentPartial

Synopsis

```
int PmPresentPartial (PmPartialSet aPS, int index)
```

Returns 1 if partial index is present in PartialSet aPS, 0 otherwise.

4.4.10 PmUnwrapPhase

Synopsis

```
float PmUnwrapPhase (float prev_freq, float next_freq, float prev_phase, float
next_phase, float dtime)
```

Returns the unwrapped phase variation of a variable-frequency sinusoid between two times prev_time and next_time.

prev_freq : frequency at prev_time

next_freq : frequency at next_time

prev_phase : phase at prev_time

next_phase : phase at next_time

dtime : [next_time - prev_time]

4.4.11 PmWrapPhase

Synopsis

```
float PmWrapPhase (float phase)
```

Wraps phase to the inverval [-PI;PI].

4.4.12 PmFreqTransp

Synopsis

```
int PmFreqTransp (PmPartialSet nextOrig, PmPartialSet nextTransf, PmPartialSet
prevOrig, PmPartialSet prevTransf, float nextFactor, float prevFactor)
```

[30] see section 4.4.7 page 17

Transposes frequencies of PartialSet nextOrig by factor nextFactor and puts the result in nextTransf.

As phase evolution is taken into account, this algorithm needs past and current data, defined at times prev_time and next_time respectively:

nextOrig : input non-transformed PartialSet at next_time.

nextTransf : output transformed PartialSet at next_time.

prevOrig : input non-transformed PartialSet at prev_time.

prevTransf : input transformed PartialSet at prev_time.

nextFactor : transposition factor at next_time.

prevFactor : transposition factor at prev_time.

If prevOrig or prevTransf are set to NULL, phase is not considered (and prevFactor is not used). In this case, phase values after frequency transposition are nonsense.

The algorithm still works if nextOrig is equal to nextTransf. In this case, nextOrig non-transformed data is overwritten.

Returns

1 if transposition was successful.

-1 if there is no available memory to perform the operation.

4.4.13 PmIvarFreqTransp

Synopsis

```
int PmIvarFreqTransp (PmPartialSet nextOrig, PmPartialSet nextTransf,
PmPartialSet prevOrig, PmPartialSet prevTransf, PmBreakPoint nextIvar, float
prevIvar)
```

Index-variable frequency transposition: transposes frequencies of PartialSet nextOrig by a factor which depends on the partial index.

Transposition factor is defined in a breakpoint function nextIvar which contains points [index ; factor].

prevIvar is the breakpoint function which has been used at prev_time.

Other features are the same as for PmFreqTransp[31].

Returns

1 if transposition was successful.

-1 if there is no available memory to perform the operation.

4.4.14 PmFvarFreqTransp

Synopsis

```
int PmFvarFreqTransp (PmPartialSet nextOrig, PmPartialSet nextTransf,
PmPartialSet prevOrig, PmPartialSet prevTransf, PmBreakPoint nextIvar, float
prevIvar)
```

[31] see section 4.4.12 page 18

Frequency-variable frequency transposition: transposes frequencies of PartialSet nextOrig by a factor which depends on the partial frequency.

Transposition factor is defined in a breakpoint function nextIvar which contains points [frequency ; factor].

prevIvar is the breakpoint function which has been used at prev_time.

Other features are the same as for PmFreqTransp[32].

Returns

1 if transposition was successful.

-1 if there is no available memory to perform the operation.

4.4.15 PmTimeScaling

Synopsis

```
int PmTimeScaling (PmPartialSet nextOrig, PmPartialSet nextTransf, PmPartialSet
prevOrig, PmPartialSet prevTransf, float scaleFactor)
```

Scales time of PartialSet nextOrig according to factor scaleFactor and puts the result in nextTransf.

The interval between nextTransf and prevTransf is calculated as [next_time-prev_time] multiplied by factor scaleFactor.

Input and output PartialSets are the same as for PmFreqTransp[33].

If prevOrig or prevTransf are set to NULL:

- phase is not considered, so phase values after time scaling are nonsense.
- the scaled time of nextTransf is obtained by multiplying the absolute time of nextOrig by the scaling factor scaleFactor. This imposes the condition

  $$prev\_factor * prev\_time <= next\_factor * next\_time$$

  in order to avoid time reversing.

Returns

1 if scaling was successful.

-1 if there is no available memory to perform the operation.

4.4.16 PmInterpPartialSet

Synopsis

```
void PmInterpPartialSet (PmPartialSet left, PmPartialSet right, PmPartialSet
interp, int phase_flag)
```

Calculates an interpolated PartialSet interp between PartialSets left and right.

[32]see section 4.4.12 page 18
[33]see section 4.4.12 page 18

The interpolation factor is calculated as

$$(interp\_time - left\_time)/(right\_time - left\_time)$$

so time of PartialSet interp must be set before calling the function.

Amplitudes are linearly interpolated.

If phase_flag is set to 0, phases are not considered and frequencies are linearly interpolated.

If phase_flag is different from 0, phases are assumed to be valid; in this case phases and frequencies are interpolated using 3rd-degree and 2nd-degree polynoms respectively.

Returns

1 if interpolation was successful.

-1 if there is no available memory to perform the operation.

4.4.17 PmSetSpectralEnv

Synopsis

```
int PmSetSpectralEnv (PmPartialSet inPS, PmPartialSet outPS, float *env, float
scale_factor, float srate, int env_size, int energy_flag)
```

Imposes the spectral envelope defined by env to inPS.

Amplitudes of partials in inPS are recalculated by interpolating env at the corresponding partial frequencies.

The spectral envelope defined by env can be previously rescaled by a scale_factor.

scale_factor > 1 stretches the envelope.

scale_factor < 1 compresses the envelope.

The result is placed in outPS, which can be eventually equal to inPS.

env is a spectral envelope defined at env_size regularly spaced points, corresponding to the frequency interval [0 - srate/2].

If energy_flag is different from 0, output and input energies are equalized.

srate : sampling rate

Returns

1 if the operation was successful.

-1 if there is no available memory to perform the operation.

4.4.18 PmMultSpectralEnv

Synopsis

```
int PmMultSpectralEnv (PmPartialSet inPS, PmPartialSet outPS, float *env, float
scale_factor, float srate, int env_size, int energy_flag)
```

Same as PmSetSpectralEnv[34] but multiplies amplitudes of partials by the envelope instead of imposing

[34] see section 4.4.17 page 21

it.

4.4.19 PmCopySubPartialSet

Synopsis

```
int PmCopySubPartialSet (PmPartialSet inPS, PmPartialSet outPS, int
*subset_indexes, int nb_indexes)
```

Partials defined in inPS, and whose indexes are present in subset_indexes, are copied to outPS.

outPS is reset to an empty PartialSet before copy is performed.

nb_indexes : number of elements of subset_indexes.

Returns

1 if the operation was successful.

-1 if there is no available memory to perform the operation.

4.4.20 PmMoveSubPartialSet

Synopsis

```
int PmMoveSubPartialSet (PmPartialSet inPS, PmPartialSet outPS, int
*subset_indexes, int nb_indexes)
```

Partials defined in inPS, and whose indexes are present in subset_indexes, are copied to outPS and removed from inPS.

outPS is reset to an empty PartialSet before move is performed.

nb_indexes : number of elements of subset_indexes.

Returns
1 if the operation was successful,
-1 if there is no available memory to perform the operation.

4.4.21 PmAddSynt

Synopsis

```
int PmAddSynt(PmPartialSetleft, PmPartialSetright, float *outSig, int maxsize,
int done, float srate, int phase_flag)
```

Performs additive synthesis in the interval between PartialSets left and right, and puts the resulting signal in outSig. At most maxsize samples are computed. Because the interval can be larger than maxsize (the output buffer size), PmAddSynt must be called in a loop:

```
    sig_done = 0;
    do {
sig_size = PmAddSynt(leftPartialSet, rightPartialSet,
     outSig, maxsize, sig_done, srate, phaseFlag);
sig_done += sig_size;

/* do something useful with the synthesized signal */
...
    } while (sig_size == maxsize);
```

If phase_flag is set to 0, phases are not considered and frequencies are linearly interpolated.

If phase_flag is different from 0, phases are assumed to be valid; in this case phases and frequencies are interpolated using 3rd-degree and 2nd-degree polynoms respectively.

srate : output sampling rate

Returns
the number of output samples actually computed if the operation was successful.
-1 if there is no available memory to perform the operation.

4.5 PartialSet Data Access Functions

4.5.1 PmPutPartialSetTime

Synopsis
```
void PmPutPartialSetTime (PmPartialSet aPS, float time)
```

4.5.2 PmGetPartialSetTime

Synopsis
```
float PmGetPartialSetTime (PmPartialSet aPS)
```

4.5.3 PmPutPartialSetNb

Synopsis
```
void PmPutPartialSetNb (PmPartialSet aPS, int nb)
```

4.5.4 PmGetPartialSetNb

Synopsis
```
int PmGetPartialSetNb (PmPartialSet aPS)
```

4.5.5 PmGetPartialSetAllocSize

Synopsis
```
int PmGetPartialSetAllocSize (PmPartialSet aPS)
```

4.5.6 PmPutPartialIndex

Synopsis
```
void PmPutPartialIndex (PmPartialSet aPS, int position, int index)
```

4.5.7 PmGetPartialIndex

Synopsis
```
int PmGetPartialIndex (PmPartialSet aPS, int position)
```

4.5.8 PmPutPartialFreq

Synopsis

```
void PmPutPartialFreq (PmPartialSet aPS, int index, float freq)
```

4.5.9 PmGetPartialFreq

Synopsis

```
float PmGetPartialFreq (PmPartialSet aPS, int index)
```

4.5.10 PmPutPartialAmpl

Synopsis

```
void PmPutPartialAmpl (PmPartialSet aPS, int index, float ampl)
```

4.5.11 PmGetPartialAmpl

Synopsis

```
float PmGetPartialAmpl (PmPartialSet aPS, int index)
```

4.5.12 PmPutPartialPhase

Synopsis

```
void PmPutPartialPhase (PmPartialSet aPS, int index, float phase)
```

4.5.13 PmGetPartialPhase

Synopsis

```
float PmGetPartialPhase (PmPartialSet aPS, int index)
```

4.5.14 PmAddPartial

Synopsis

```
int PmAddPartial (PmPartialSet aPS, int index, float freq, float ampl, float
phase)
```

Adds a new partial with index index to PartialSet aPS.

freq, ampl and phase are the parameters of the new partial.

Returns

1 if the new partial was added.

0 if a partial with index index was already present in PartialSet aPS. In this case, new parameters do not overwrite old ones.

4.5.15 PmDeletePartial

Synopsis

```
int PmDeletePartial (PmPartialSet aPS, int index)
```

Deletes partial with index index from PartialSet aPS.

Returns

1 if the partial was deleted.

0 if a partial with index index did not exist in PartialSet aPS.

4.5.16 PmDeleteSubPartialSet

Synopsis

```
int PmDeleteSubPartialSet (PmPartialSet aPS, int *subset_indexes, int nb_indexes)
```

Partials defined in aPS, and whose indexes are present in subset_indexes, are removed from aPS.

Returns the number of deleted partials.

4.6 Frame Sets

4.6.1 PmCreateFrameSet

Synopsis

```
PmFrameSet PmCreateFrameSet (int nb_of_frames, int nb_of_paramsets)
```

Creates a new FrameSet.

nb_of_frames : max number of frames of the FrameSet.

nb_of_paramsets : max number of ParamSets the FrameSet can support.

Returns a new FrameSet, or NULL if there is no available memory.

4.6.2 PmReallocFrameSet

Synopsis

```
PmFrameSet PmReallocFrameSet (PmFrameSet aFS, int nb_of_frames, int
nb_of_paramsets)
```

Reallocates FrameSet aFS in the two dimensions: frames and sets of parameters.

nb_of_frames : new max number of frames of the FrameSet.

nb_of_paramsets : new max number of ParamSets the FrameSet can support. If a new dimension size (i.e. nb_of_frames or nb_of_paramsets) is smaller than the current dimension size, reallocation of this dimension is ignored.

Returns

If reallocation is successful, returns the same input FrameSet aFS with reallocated number of frames and ParamSets.

If reallocation fails, returns NULL and the size of aFS remains unchanged.

If aFS is NULL, returns a new FrameSet or NULL if reallocation failed.

Reallocation is performed in two stages: first for ParamSets, secondly for Frames. It can eventually fail at the second stage; in this case reallocation of ParamSets remains valid even if the function returns NULL.

Most functions that operate on FrameSets provide for automatic reallocation.

4.6.3 PmFreeFrameSet

Synopsis

```
void PmFreeFrameSet (PmFrameSet aFS)
```

Frees FrameSet aFS.

4.6.4 PmCreateParamSet

Synopsis

```
PmParamSetId PmCreateParamSet (PmFrameSet aFS, int paramset_length, int
paramset_type, char *paramset_name)
```

Allocates memory for a set of paramset_length elements of paramset_type type in each frame of FrameSet aFS.

paramset_type possible values are:

C basic data types:

```
PM_CHAR, PM_SHORT, PM_INT, PM_LONG, PM_FLOAT, PM_DOUBLE
```

Pm specific data types:

```
PM_PARTIAL
```

Returns an identifier of the new ParamSet, or -1 if there is no available memory to create the new paramset, or if paramset_type is unknown for the library.

4.6.5 PmFreeParamSet

Synopsis

```
void PmFreeParamSet (PmFrameSet aFS, PmParamSetId aParamSetId)
```

Frees ParamSet identified by aParamSetId in FrameSet aFS.

4.7 FrameSet Data Access

4.7.1 PmGetNbOfFrames

Synopsis

```
int PmGetNbOfFrames (PmFrameSet aFS)
```

Returns the number of active frames in FrameSet aFS.

4.7.2 PmDecrNbOfFrames

Synopsis

```
void PmDecrNbOfFrames (PmFrameSet aFS, int N)
```

Decrease in N the number of active frames of FrameSet aFS, by throwing away the last N frames.

N is automatically limited to the number of active frames of FrameSet aFS.

4.7.3 PmGetBeginTime

Synopsis

```
float PmGetBeginTime (PmFrameSet aFS)
```

Returns the begin time of FrameSet aFS.

4.7.4 PmPutBeginTime

Synopsis

```
void PmPutBeginTime (PmFrameSet aFS, float time)
```

Sets to time the begin time of FrameSet aFS.

4.7.5 PmGetEndTime

Synopsis

```
float PmGetEndTime (PmFrameSet aFS)
```

Returns the end time of FrameSet aFS.

4.7.6 PmPutEndTime

Synopsis

```
void PmPutEndTime (PmFrameSet aFS, float time)
```

Sets to time the end time of FrameSet aFS.

4.7.7 PmGetNbOfParamSets

Synopsis

```
int PmGetNbOfParamSets (PmFrameSet aFS)
```

Returns the number of active ParamSets in FrameSet aFS.

4.7.8 PmGetActiveParamSetId

Synopsis

```
PmParamSetId PmGetActiveParamSetId (PmFrameSet aFS, int pos)
```

Returns the identifier of the ParamSet in position pos in FrameSet aFS.

4.7.9 PmGetParamSetType

Synopsis

```
int PmGetParamSetType (PmFrameSet aFS, PmParamSetId aParamSetId)
```

Returns the type of ParamSet identified by aParamSetId in FrameSet aFS.

ParamSet type is one of the following:

C basic data types:

```
PM_CHAR, PM_SHORT, PM_INT, PM_LONG, PM_FLOAT, PM_DOUBLE
```

Pm specific data types:

```
PM_PARTIAL
```

4.7.10 PmGetParamSetName

Synopsis

```
char * PmGetParamSetName (PmFrameSet aFS, PmParamSetId aParamSetId)
```

Returns the name of ParamSet identified by aParamSetId in FrameSet aFS.

4.7.11 PmGetParamSetElemSize

Synopsis

```
int PmGetParamSetElemSize (PmFrameSet aFS, PmParamSetId aParamSetId)
```

Returns the size of the variable type of ParamSet identified by aParamSetId in FrameSet aFS.

4.8 Frame Data Access

4.8.1 PmGetFrameTime

Synopsis

```
float PmGetFrameTime (PmFrameSet aFS, int frame_index)
```

Returns the time of frame in position frame_index in FrameSet aFS.

Returns PM_MAXFLOAT if frame_index is equal to or greater than the number of active frames in FrameSet aFS.

4.8.2 PmPutFrameTime

Synopsis

```
void PmPutFrameTime (PmFrameSet aFS, int frame_index, float time)
```

Sets to time the time of frame in position frame_index in FrameSet aFS.

4.8.3 PmGetParamSetAllocSize

Synopsis

```
int PmGetParamSetAllocSize (PmFrameSet aFS, int frame_index, PmParamSetId
aParamSetId)
```

Returns the allocated number of elements, in frame number frame_index, for a ParamSet identified by aParamSetId in FrameSet aFS,

-1 if frame_index is equal to or greater than the number of allocated frames in FrameSet aFS.

4.8.4 PmGetParamSetNb

Synopsis

```
int PmGetParamSetNb (PmFrameSet aFS, int frame_index, PmParamSetId aParamSetId)
```

Returns the current number of elements, in frame number frame_index, of a ParamSet identified by aParamSetId in FrameSet aFS.

-1 if frame_index is equal to or greater than the number of active frames in FrameSet aFS.

4.8.5 PmGetParamSetPtr

Synopsis

```
void * PmGetParamSetPtr (PmFrameSet aFS, int frame_index, PmParamSetId
aParamSetId)
```

Returns the data pointer of ParamSet identified by aParamSetId in FrameSet aFS, in frame number frame_index.

NULL if frame_index is equal to or greater than the number of allocated frames in FrameSet aFS.

4.9 FrameSet Data Transfer

4.9.1 PmGetParamSet

Synopsis

```
int PmGetParamSet (PmFrameSet aFS, int frame_index, PmParamSetId aParamSetId,
void data_ptr, int data_nb)
```

Copies a ParamSet from frame frame_index to the space pointed by data_ptr.

The ParamSet is identified by aParamSetId in FrameSet aFS.

If the ParamSet lenght in frame frame_index is greater than argument data_nb, data_nb elements are copied.

Otherwise, the entire ParamSet is copied.

Returns

0 if frame_index is equal to or greater than the number of active frames.

1 if operation is successful.

4.9.2 PmPutParamSet

Synopsis

```
int PmPutParamSet (PmFrameSet aFS, int frame_index, PmParamSetId aParamSetId,
void data_ptr, int data_nb)
```

Copies data_nb elements from the space pointed by data_ptr to a ParamSet in frame frame_index.

The ParamSet is identified by aParamSetId in FrameSet aFS.

data_ptr is assumed to point at data of the same type as the ParamSet.

Returns

0 if frame_index is equal to or greater than the number of active frames.

1 if operation is successful.

4.9.3 PmAddFrameData

Synopsis

```
int PmAddFrameData (PmFrameSet aFS, float time, PmParamSetId *theParamSetIds,
void **theDataPtrs, int *theNbOfElems, int nbOfParamSets)
```

Fills a new frame of data and increments by one the number of active frames of FrameSet aFS.

time is the time to be assigned to the new frame.

Each line of vectors theParamSetIds, theDataPtrs and theNbOfElems correspond to a source data set and its corresponding destination ParamSet in FrameSet aFS.

theParamSetIds is a vector of destination ParamSet identifiers.

ParamSets identified by theParamSetIds should have been created previously. If an identifier in theParamSetIds is not listed among existing ParamSets, the corresponding data set is not transferred.

theDataPtrs is a vector of pointers to the source data sets.

It is assumed that each source data set and its corresponding destination ParamSet have the same variable type.

theNbOfElems is a vector specifying the number of elements of each source data set.

nbOfParamSets is the number of transferred data sets.

Returns

[number of transferred data sets] if operation is successful.

-1 if there is no available memory to perform the operation.

4.9.4 PmGetFrameData

Synopsis

```
int PmGetFrameData (PmFrameSet aFS, int frame_index, float *ptime, PmParamSetId
*theParamSetIds, void **theDataPtrs, int *theNbOfElems, int nbOfParamSets)
```

Retrieves data from ParamSets in frame number frame_index of FrameSet aFS.

Does not remove frame number frame_index from FrameSet aFS.

The contents of pointer ptime will be filled with the time value of frame number frame_index.

If ptime is NULL, no time value is transferred.

Each line of vectors theParamSetIds, theDataPtrs and theNbOfElems correspond to a destination data set and its corresponding source ParamSet in FrameSet aFS.

theParamSetIds is a vector of source ParamSet identifiers.

If an identifier in theParamSetIds is not listed among existing ParamSets, the corresponding transfer is skipped.

theDataPtrs is a vector of pointers to the destination data sets.

It is assumed that each destination data set and its corresponding source ParamSet have the same variable type.

theNbOfElems is a vector specifying the length of each destination data set.

The number of transferred data is limited to the destination data set length if this one is shorter than the corresponding ParamSet.

Otherwise, the entire ParamSet is copied.

nbOfParamSets is the number of transferred ParamSets.

Returns

[number of transferred ParamSets] if operation is successful.

0 if frame_index is equal to or greater than the number of active frames in FrameSet aFS.

4.10 FrameSet Editing

4.10.1 PmCutFrameSet

Synopsis

```
PmFrameSet PmCutFrameSet (PmFrameSet aFS, float tInf, float tSup)
```

Removes the set of frames in the interval [tInf;tSup], and connects the extremities of the remaining portions of FrameSet aFS.

Returns a new FrameSet which contains the removed set of frames.

NULL if there is no frame in the interval [tInf;tSup], or if there is no available memory to perform the operation.

4.10.2 PmInsertFrameSet

Synopsis

```
void PmInsertFrameSet (PmFrameSet aFS, PmFrameSet insFS, float tIns)
```

Inserts FrameSet insFS at time tIns of FrameSet aFS.

Returns

1 if insertion is successful.

0 if insertion time tIns is outside the definition interval of FrameSet aFS.

-1 if there is no available memory to perform the operation.

4.10.3 PmCopyFrameSet

Synopsis

```
PmFrameSet PmCopyFrameSet (PmFrameSet aFS, float tInf, float tSup)
```

Copies the set of frames in the interval [tInf;tSup] of FrameSet aFS.

Returns a new FrameSet which contains the copied set of frames.

NULL if there is no frame in the interval [tInf;tSup], or if there is no available memory to perform the operation.

4.10.4 PmShiftFrameSet

Synopsis

```
int PmShiftFrameSet (PmFrameSet aFS, int nb_of_frames)
```

Shifts to the left nb_of_frames frames of FrameSet aFS.

nb_of_frames is automatically limited to the number of active frames of FrameSet aFS.

4.10.5 PmReverseFrameSet

Synopsis

```
void PmReverseFrameSet (PmFrameSet aFS, float tInf, float tSup)
```

Reverses the set of frames in the interval [tInf;tSup] of FrameSet aFS.

4.11 PartialSet Processing at FrameSet Level

4.11.1 PmFadeHarmEnds

Synopsis

```
int PmFadeHarmEnds (PmFrameSet aFS, PmParamSetId aParamSetId, float tAttack,
float tRelease, int input_flag)
```

Smooth trajectories of partials in ParamSet aParamSetId of FrameSet aFS.

Amplitude fading is made at trajectory birth and death ends.

Frequency in the fading portion is kept constant at the corresponding end value.

tAttack: fade in duration.

tRelease: fade out duration.

For amplitude fading to be performed, a partial must remain dead for a time of at least (tRelease + tAttack). Otherwise amplitude and frequency linear interpolation is done in between death and birth ends.

It is assumed that a given index identifies always the same partial trajectory all along FrameSet aFS.

This is always the case when partials are harmonic.

Instead, for inharmonic partials, a partial with index i which dies at time t may have no relation at all with another partial with same index i which is born at a future time $t+dt$. If $dt <$ (tAttack+tRelease), PmFadeHarmEnds will join the two independent partials by interpolation in between death and birth ends.

Returns

A minimum duration of (tRelease + tAttack + tRelease) is required for FrameSet aFS. As time step between frames may be variable, the number of frames required to satisfy this condition may be variable too.

While FrameSet aFS is too short, PmFadeHarmEnds does nothing and returns 0, waiting for a longer FrameSet.

If FrameSet aFS is long enough, PmFadeHarmEnds performs smoothing and returns a positive integer, equal to the number of processed frames that can be shifted out of FrameSet aFS using PmShiftFrameSet.

If input_flag is set to 0, PmFadeHarmEnds does not expect for new frames to be added to FrameSet aFS and finishes smoothing regardless of the FrameSet duration.

PmFadeHarmEnds returns -1 if aParamSetId does not correspond to a ParamSet of type PM_PARTIAL or if there is no available memory to perform the operation.

4.11.2 PmScaleFrameSetTime

Synopsis

```
PmFrameSet PmScaleFrameSetTime (PmFrameSet aFS, PmBreakPoint aBP)
```

Scales time of FrameSet aFS according to a breakpoint function aBP specifying output time versus input time.

Returns a new scaled FrameSet.

NULL if there is no available memory to perform the operation.

---

5 Programs

5.1 pmconvert, formattosdif, fmttosdif, sdiftoformat, sdiftofmt, formattofmt, fmttoformat, picstosdif, picsbintosdif

The *pmconvert* program converts partial data between the formats ascii (.format), binary (.fmt), and SDIF[35] (.sdif).

Synopsis

```
usage: pmconvert [options] [infile [outfile]]

pmconvert converts between files with partials (.format, .fmt, .sdif),
peaks (.pics), and break-point functions (.f0).
Files can be '-' for standard input/output.

options:
        -q      work quietly
        -R <sr> sampling rate for peaks conversion
        -h      this help
```

When called as `pmconvert`, the input and output files must be given, and their types are inferred from the extensions. I.e. you cannot use standard input/output.

The programs *formattosdif, fmttosdif, sdiftoformat, sdiftofmt, formattofmt, fmttoformat, picstosdif, picsbintosdif* are copies of *pmconvert*, which take the input and output file types from their name. Here, you can use standard input/output.

Note that you can use *gzip*-compressed files (.gz) as input/output, which are automatically decompressed/compressed.

5.2 The pm Parser

...performs additive analysis/synthesis

[35] `http://www.ircam.fr/sdif`

Synopsis

```
pm [options...] [output file]

Options:
-v <Verbose>
-S <Input FileName>          (.fft or .format)
-P <Fundamuntal FileName>    (.F0)
-A <Action>                  (par:PartialFollow, syn:Synthesis)
-O <Partial Type>            (a:ASCII, b:Binary, s:SDIF)
-q <Number Of Partials>      (all by default)
-M <Window Analysis Size>    (nbSamples)
-N <Window FFT Size>         (nbSamples)
-I <Window Analysis Step>    (nbSamples)
-W <Analysis Type>
-p <Synthesis With Phase>
-R <Sampling Rate>           (nbSamples/s)
-B <Begin>                   (time)
-E <End>                     (time)
-c <Bandwidth Partial Seeve> (coeff 0<k<1)
-a <Envelopp Attack>         (time)
-r <Envelopp Release>        (time)
```