Frame Recover Library 1.0.0 Manual

(February 2005)

R. Bernardini (bernardini@uniud.it)

Table of Contents

1 Fast Overview

Frame recover library is a `C++` library for recovering losses in coefficient streams obtained by means of an oversampled filter bank. This chapter describes briefly the goals of this library, what it provides and the application software which comes with it.

1.1 The goal of the library

This library implements the algorithm of [???] for coefficient recovering for frame-analyzed signals.

Frame analysis (e.g. analysis with a redundant filter bank) is a recently proposed solution to the problem of transmitting in a robust way multimedia signals over unrielable channels. Here and in the following I will suppose that you are confident with the idea of robust transmission and oversampled filter banks. If the first sentence of this paragraph made no sense to you, then almost surely you do not need this library and you can stop reading here.

This library was designed by trying to meet the following apparently incompatible goals

- The library must be *flexible* enough to be used in several context:
  - with one-dimensional or multi-dimensional signals
  - with signals padded with zeros or periodically repeated
  - with coefficients stored in a file, in memory or coming from a network interface
  - with real, complex or even rational coefficients
  - even with frame not associated with an oversampled filter bank.
- Flexibility must not be obtained at the expense of a complex interface. It should be possible to *apply the library to the most common cases* (real coefficients obtained by means of an oversampled filter bank and stored in a file or in memory) *with very little work*.
- It should be easy to modify the library in order to suit it to one's own desires.

Those goals were met by interfacing the main computational function (`recover_stream`) to the "external world" by means of *abstract class* `StreamDescr` whose objects takes care of coefficient I/O and carry all the informations necessary to `recover_stream`. In order to actually use `recover_stream` the user should write a derived class of `StreamDescr` suited to its own needs. Since the filter bank based case is the most common case, the library gives a *pret-a-porter* class `FilterBankStream` which allows one to use `recover_stream` with almost no work at all. (see Section 1.2 [Simple Example], page 1).

1.2 Some usage examples

In this section we show some simple examples of usage of `recover_stream` together with `FilterBankStream`. We cover the four most common cases.

1.2.1 File to File example

In this example we suppose that the input coefficients are stored on a file according to the format described in Section B.1 [Coefficient Format], page 17. A brief description of the program follows.

```
#include<iostream>
#include"recover.h"
#include"filter_bank_stream.h"

using namespace std;

void die(string msg)
{ cerr << msg << endl;  exit(1); }

int main(int argc, char **argv)
{
  //
  // Command line parameters processing
  //
  if (argc < 4)
      die("Usage: test_recover sp_table coeff_in coeff_out\n");

  string sp_filename(argv[1]);
  string in_filename(argv[2]);
  string out_filename(argv[3]);

  //
  // Load the scalar product table
  //
  SpTable<double> sp;
  load_sp_table(sp, sp_filename);

  //
  // Create the data descriptor with the two channels and the
  // scalar product table
  //
  FilterBankStream fbstream(in_filename,       // Input
                            out_filename,      // Output
                            sp);               // SP table

  //
  // Recover the lost coefficients.
  //
  recover_stream(fbstream);

  exit(0);
}
```

A brief comment about the program is in order. A `FilterBankStream` needs to know

1. Some signal parameters such as the signal size and how the signal was padded (with zeros or periodically repeated)
2. The number of channels of the analysis filter bank

3. The scalar products between the frame functions and the dual ones.

When the coefficients are read from a file the first two informations are read from the file header (see Section B.1 [Coefficient Format], page 17) and the third one can be obtained from the table of scalar products. Because of this, we need to give to the `FilterBankStream` constructor at line 32 only the names of the input and output files and the scalar product table loaded at line 26.

When the `FilterBankStream` is ready, we can give it to `recover_stream` (line 39) and let it do its work.

1.2.2 Memory to File example

In this example we suppose that the coefficient are already in memory (for example, they are the result of some other processing) and we want to write the recovered stream to a file.

```
#include<iostream>
#include"recover.h"
#include"filter_bank_stream.h"

using namespace std;

void die(string msg)
{ cerr << msg << endl;  exit(1); }

int main(int argc, char **argv)
{
  vector<double> val;      // Coefficient values
  vector<bool>   known;    // known[n]==true iff n-th coefficient arrived

  vector<unsigned int>signal_size;    // Signal dimensions
  vector<border_type> padding_type;   // How the signal was padded


  //
  // Command line parameters processing
  //
  if (argc < 3)
      die("Usage: test_recover sp_table coeff_out\n");

  string sp_filename(argv[1]);
  string out_filename(argv[2]);

  //
  // Load the scalar product table
  //
  SpTable<double> sp;
  load_sp_table(sp, sp_filename);

  //
```

```
35:   // Get the coefficient values and the signal size
36:   //
37:
38:    ...here you should put the code which reads the signal size,...
39:    ...how the signal was padded and the coefficient values     ...
40
41    //
42:   // Create the data descriptor with the two channels and the
43:   // scalar product table
44:   //
45:
46:   FilterBankStream fbstream(val, known,        // Input
47:                             out_filename,      // Output
48:                             sp,                // SP table
49:                             signal_size,       // Signal size
50:                             padding_type);     // Signal padding
51:
52:   //
53:   // Recover the lost coefficients.
54:   //
55:   recover_stream(fbstream);
56:
57:   exit(0);
58: }
```

This case is sligthly more complex than the example shown in Section 1.2.1 [File to File], page 1 because now the signal dimensions and padding cannot be read from file, but they must be given explicitely to the `FilterBankStream` constructor (lines 46–50) via arrays `signal_size` and `padding_type`.

The input coefficients are given to the `FilterBankStream` constructor via two arrays: the first one (`val`) contains the coefficient values ordered according to the same order used in the coefficient file format (see Section B.1 [Coefficient Format], page 17), while the second arrays (`known`) is a vector of `bool` such that `known[n]` is `true` if and only if the `n`-th coefficient arrived. If `known[n]` is `false`, `val[n]` is not read and it can have any value.

1.2.3 File to Memory example

This case is the "dual" of the case considered in Section 1.2.2 [Memory to File], page 3: the input coefficients are stored in a file (still according to the format described in Section B.1 [Coefficient Format], page 17) and the recovered stream will be in memory.

```
#include<iostream>
#include"recover.h"
#include"filter_bank_stream.h"

using namespace std;

void die(string msg)
{ cerr << msg << endl;  exit(1); }
```

```

int main(int argc, char **argv)
{
  //
  // Command line parameters processing
  //
  if (argc < 4)
      die("Usage: test_recover sp_table coeff_in coeff_out\n");

  string sp_filename(argv[1]);
  string in_filename(argv[2]);
  string out_filename(argv[3]);

  //
  // Load the scalar product table
  //
  SpTable<double> sp;
  load_sp_table(sp, sp_filename);

  //
  // Create the data descriptor with the two channels and the
  // scalar product table
  //
  vector<double> val;
  vector<bool>   known;
  FilterBankStream fbstream(in_filename,       // Input
                            val, known,        // Output
                            sp);               // SP table

  //
  // Recover the lost coefficients.
  //
  recover_stream(fbstream);

  exit(0);
}
```

Since in this case the input coefficients are stored in a file, it is not necessary to give to the `FilterBankStream` constructor the informations about the signal size and the padding type.

The output coefficients will be stored in two arrays (`val` and `known`) according to the same convention used in the "Memory to File" (see Section 1.2.2 [Memory to File], page 3) case. Note that even after the call to `recover_stream` some element of `known` can still be false if recovering was not possible because of excessive losses.

1.2.4 Memory to Memory example

Finally, we consider the "memory to memory" case. The same comments used for the other cases (Section 1.2.1 [File to File], page 1, Section 1.2.3 [File to Memory], page 4, Section 1.2.2 [Memory to File], page 3) apply. The only new observation is that this time the same pair of array (`val` and `known`) is used both for input and for output. If desired, it is also possible to use different vectors for the original coefficients and for the recovered ones.

```
#include<iostream>
#include"recover.h"
#include"filter_bank_stream.h"

using namespace std;

void die(string msg)
{ cerr << msg << endl;  exit(1); }

int main(int argc, char **argv)
{
  //
  // Command line parameters processing
  //
  if (argc < 4)
      die("Usage: test_recover sp_table coeff_in coeff_out\n");

  string sp_filename(argv[1]);
  string in_filename(argv[2]);
  string out_filename(argv[3]);

  //
  // Load the scalar product table
  //
  SpTable<double> sp;
  load_sp_table(sp, sp_filename);

  //
  // Create the data descriptor with the two channels and the
  // scalar product table
  //
  vector<double> val;
  vector<bool>   known;
  FilterBankStream fbstream(val, known,        // Input and Output
                            sp,                // SP table
                            signal_size, ext); // Signal size

  //
  // Recover the lost coefficients.
```

```
  //
  recover_stream(fbstream);

  exit(0);
}
```

2 The library

2.1 The procedure `recover_stream()`

Procedure `recover_stream()` is the "computational core" of the library and implements the algorithm of [???] in complete generality. In order to do its work `recover_stream()` needs to

- read the input stream of coefficients and write the restored one;
- know which coefficients are neighbors of a given one (see Section 2.3 [Internals], page 12 and [???])
- know the scalar products between frame and dual functions (see Section 2.3 [Internals], page 12 and [???])

All those informations are stored in a *stream descriptor* passed as parameter to `recover_stream()`. In `C++` a *stream descriptor* is represented by an object of class `StreamDescr`.

Actually, since the informations above strongly depend on how the coefficients were generated and how they are stored and/or transmitted, class `StreamDescr` is just an *abstract class* whose only goal is to describe the interface that a "good" *stream descriptor* should implement (see Section 2.3.1 [StreamDescr interface], page 12 for details).

This implies that it is not possible to allocate objects of type `StreamDescr`, but the user needs to create a new class, derived from `StreamDescr` and suited to its own needs. Note that since the methods of `StreamDescr` are declared `virtual` new stream descriptors can be used without the need of recompiling the library.

The use of an object derived from an abstract class as interface between `recover_stream` and the "external world" gives great flexibility to the library, at the expense of requiring the user to write its own stream descriptor.

Since the most common case is the case of a coefficient stream stored on disk or in memory and obtained by means of an oversampled filter bank, the library provides a *pret-a-porter* stream descriptor `FilterBankStream` suited to this case (see Section 2.2 [FilterBankStream], page 8).

2.2 Stream Descriptor `FilterBankStream`

2.2.1 What a `FilterBankStream` is

`FilterBankStream` is a `StreamDescr` especially suited for the case of coefficients obtained by means of an oversampled filter bank.

2.2.2 Interface of `FilterBankStream`

The only user methods of `FilterBankStream` are the constructors described in the following. `FilterBankStream` has several different constructors, one for each possible case of interest. In order to understand the constructor interface, observe that a `FilterBankStream` must know

1. the coefficient source
2. the coefficient destination

3. the scalar products between frame and dual functions
4. signal structure informations such as
   - the number of channels of the filter bank
   - the size of the signal
   - how the input signal was extended before filtering (with zeros or by periodicity)

In general, the first and second informations are given to the `FilterBankStream` via two objects of type `CoeffSource`. A `CoeffSource` models a stream of coefficients which can be sequentially read/write. At each read access the `CoeffSource` returns if the next coefficient has been received and its value. A `FilterBankStream` supposes that a `CoeffSource` stores the coefficients according to the order used in the coefficient files (see Section B.1 [Coefficient Format], page 17).

Note the difference between a `FilterBankStream` and a `CoeffSource`: the latter is just a sequence of coefficients with no knowledge about channels, signal dimensions and so on... Instead a `FilterBankStream` organizes the coefficients in macro-coefficients, handle the signal padding and so on. By making a parallel with databases, one could say that a `CoeffSource` represents the binary file which holds the database (it is just a sequence of byte), while a `FilterBankStream` is a higher-level interface which allows to access the database by records.

In order to have a greater generality, class `CoeffSource` is an *abstract* one which specifies the minimum interface of a source of coefficients (see Section 2.3.2 [CoeffSource interface], page 14). To enhance the usability the library includes two derived classes of `CoeffSource` for the two cases of more common interest, i.e, coefficients stored in memory (class `Array_CoeffSource` or in a file (class `File_CoeffSource`).

In order to make the library usage still easier, class `FilterBankStream` has specialized constructors which only require a filename (if the coefficients are on file) or a pair of arrays (if the coefficients are in memory).

The third information (the scalar products between frame and dual functions) is given via an object of class `SpTable` which is usually loaded with `load_sp_table` (see Section 2.3.3 [SpTable interface], page 14) as in

```
SpTable<double> sp;
load_sp_table(sp, sp_filename);
```

The fourth group of informations (signal size, signal extension and so on) can sometimes be read from the input stream (for example, if the input stream is a file, such informations can be read from the file header (see Section B.1 [Coefficient Format], page 17)). In this case is not necessary to give the informations to the constructor.

2.2.3 Available `FilterBankStream` constructors

Every `FilterBankStream` constructor follows the same convention in parameter order

1. A description of the coefficient source
2. A description of the coefficient destination (this may be not present if the restored coefficients can be rewritten back to the source)
3. The scalar product table
4. If necessary, a description of the signal size (when this information cannnot be obtained from the coefficient source)

The possible source/destinations are

1. Files. They may store signal size informations and are specified by means of their name.
2. Memory arrays. They cannot store signal size informations and are specified by means of a pair of arrays
   1. an array of `bool`s, `known`, such that `known[n]` is `true` if and only if the `n`-th coefficient arrived.
   2. an array of `double`s, `val`, such that `val[n]` is the value of the `n`-th coefficient. If `known[n]` is `false`, `val[n]` is not read and it can have any value

   For an example, See Section 1.2.2 [Memory to File], page 3.
3. Objects belonging to a class derived from the abstract class `CoeffSource`. This allows to use the class `FilterBankStream` with other setups (e.g., if the coefficients are received through a network socket).

2.2.3.1 Source and destination in memory

```
FilterBankStream(std::vector<double> &val_src,  // Input values
                 std::vector<bool>   &rec_src,  // Received?
                 std::vector<double> &val_dst,  // Output values
                 std::vector<bool>   &rec_dst,  // Restored?
                 const scal_prod_table &sp,     // SP table
                 const std::vector<unsigned int> &sz, // signal size
                 const std::vector<border_type> &extension); // padding
```

```
FilterBankStream(std::vector<double> &val_src,  // Input values
                 std::vector<bool>   &rec_src,  // Received?
                 std::vector<double> &val_dst,  // Output values
                 std::vector<bool>   &rec_dst,  // Restored?
                 const scal_prod_table &sp,     // SP table
                 const std::vector<unsigned int> &sz, // signal size
                 border_type extension);   // same padding along all
                                           // the dimensions
```

```
FilterBankStream(std::vector<double> &val_src_dst,  // Input and output
                 std::vector<bool>   &rec_src_dst,
                 const scal_prod_table &sp,         // SP table
                 const std::vector<unsigned int> &sz, // size
                 const std::vector<border_type> &extension);
```

```
FilterBankStream(std::vector<double> &val_src_dst,  // Input and output
                 std::vector<bool>   &rec_src_dst,
                 const scal_prod_table &sp,         // SP table
                 const std::vector<unsigned int> &sz,
                 border_type extension);   // same padding along all
                                           // the dimensions
```

2.2.3.2 Source and destination on file

```
FilterBankStream(const std::string &filein,    // Input
```

```
                 const std::string &fileout,   // Output
                 const scal_prod_table &sp,    // SP table
                 const std::vector<unsigned int> &sz,
                 const std::vector<border_type> &extension);
FilterBankStream(const std::string &filein,    // Input
                 const std::string &fileout,   // Output
                 const scal_prod_table &sp,    // SP table
                 const std::vector<unsigned int> &sz,
                 border_type extension);   // same padding along all
                                           // the dimensions
FilterBankStream(const std::string &filein,    // Input
                 const std::string &fileout,   // Output
                 const scal_prod_table &sp);   // SP table
                 // size and padding read from the input file
```

2.2.3.3 Source on file and destination in memory

```
FilterBankStream(const std::string &filein,     // Input
                 std::vector<double> &val_dst,  // Output
                 std::vector<bool>   &rec_dst,  // Recovered?
                 const scal_prod_table &sp,
                 const std::vector<unsigned int> &sz,
                 const std::vector<border_type> &extension);
FilterBankStream(const std::string &filein,     // Input
                 std::vector<double> &val_dst,  // Output
                 std::vector<bool>   &rec_dst,  // Recovered?
                 const scal_prod_table &sp,
                 const std::vector<unsigned int> &sz,
                 border_type extension);   // same padding along all
                                           // the dimensions
FilterBankStream(const std::string &filein,     // Input
                 std::vector<double> &val_dst,  // Output
                 std::vector<bool>   &rec_dst,  // Recovered?
                 const scal_prod_table &sp);
                 // size and padding read from the input file
```

2.2.3.4 Source in memory and destination on file

```
FilterBankStream(std::vector<double> &val_src, // Input
                 std::vector<bool>   &rec_src, // Received?
                 const std::string &fileout,   // Output
                 const scal_prod_table &sp,
                 const std::vector<unsigned int> &sz,
                 const std::vector<border_type> &extension);
FilterBankStream(std::vector<double> &val_src,  // Input
                 std::vector<bool>   &rec_src,  // Received?
                 const std::string &fileout,    // Output
```

```
                  const scal_prod_table &sp,
                  const std::vector<unsigned int> &sz,
                  border_type extension);   // same padding along all
                                            // the dimensions
```

2.2.3.5 Source and destination generic

```
FilterBankStream(CoeffSource &src,
                 CoeffSource &dst,
                 const scal_prod_table &sp,
                 const std::vector<unsigned int> &sz,
                 const std::vector<border_type> &ext);

FilterBankStream(CoeffSource &src,
                 CoeffSource &dst,
                 const scal_prod_table &sp,
                 const std::vector<unsigned int> &sz,
                 border_type ext);

FilterBankStream(CoeffSource &src,
                 CoeffSource &dst,
                 const scal_prod_table &sp);
```

2.3 Internal details

2.3.1 Interface of a *stream descriptor*

2.3.1.1 Overview

Class `StreamDescr` is an *abstract* class which describes the minimum interface of a *stream descriptor*. The duty of a *stream descriptor* is to give to `recover_stream()` all the necessary informations about the coefficient stream and allow for coefficient I/O.

More in details, the informations which a *stream descriptor* must make available to `recover_stream()` are

1. The value of the scalar products between frame and dual functions
2. The set of coefficients having a given *processing time* (see Appendix A [Coefficient Stream Model], page 16)
3. If the macro-coefficient corresponding to a given processing time is complete or not (see Appendix A [Coefficient Stream Model], page 16)
4. The set of processing times of the neighbors of a given coefficient
5. The *waiting-time* of a given coefficient (i.e., the maximum among the processing times of its neighbors)

A *stream descriptor* must also accept from `recover_stream()`

1. The cofficient values (both the received and the recovered ones)
2. A basis of the unrecoverable space in case of unrecoverable losses.

2.3.1.2 Formal description

Here we give a list of methods of class `StreamDescr`. You will notice that some methods are *pure virtual*, while some other are not. We decided to make a method non pure virtual if it was possible to code that method by exploiting other methods of `StreamDescr`. For example, method `receive(const std::set<Coefficient> &)`, which is used by `recover_stream()` in order to output a macro-coefficient, is not pure virtual since it can be coded by repeatedly calling `receive(const Coefficient &)` which outputs a single coefficient.

This choice allows one to derive a new class from `StreamDescr` writing only the minimum amount of code, while leaving the possibility of rewriting some of the other methods for reasons of efficiency (for example, depending on the coefficient organization, it could be more efficient to output a whole macro-coefficient instead of outputting each single coefficient).

`double sp_table(const Coefficient &r, const Coefficient &c)` **Pure virtual.**
: Returns the value of the scalar product between the dual function associated with `r` and the frame function associated with `c`.

`unsigned int size()` **Pure virtual**
: Return the total number of macro-coefficients, i.e., the one plus the maximum processing time.

`bool again(unsigned int proc_time)`
: Return true if there is a macro coefficient with processing time `proc_time`. The default behavior is to check if `proc_tim` is less than the value returned by method `size()`. It can be rewritten for the sake of efficiency.

`std::set<Coefficient> coeffs(unsigned int proc_time)` **Pure virtual.**
: Returns the set of coefficients belonging to the macro-coefficient with processing time `proc_time`

`bool is_complete(unsigned int proc_time)`
: Return true if all the coefficients belonging to the macro-coefficient with processing time `proc_time` have been received. The default behavior is to check the `known` field of all the coefficients returned by `coeffs(proc_time)`. It can be rewritten for the sake of efficiency.

`std::set<int> neighborhood(unsigned int proc_time) const` **Pure virtual.**
: Return the set of processing times of the macro-coefficients belonging to the neighborhood of the macro-coefficient with processing time `proc_time`

`int waiting_time(unsigned int proc_time) const`
: Return the waiting time of the macro-coefficient with processing time `proc_time`. The default behavior is to return the maximum among the values returned by `neighborhood(proc_time)`. It can be rewritten for the sake of efficiency.

`void receive(const Coefficient &coeff)` **Pure virtual.**
: Send to the output stream coefficient `coeff`

`void receive(const std::set<Coefficient> &macro_coeff)`
: Send to the output stream all the coefficients in the macro-coefficient `macro_coeff`. The default behavior is to repeatly call `receive(const Coefficient`

&coeff) for every element in macro_coeff. It can be rewritten for the sake of efficiency.

void kernel_vec(const std::vector<double> &val, const std::vector<Coefficient> &c)

Each time an irrecoverable loss is detected, recover_stream() calls this function in order to comunicate a basis of the kernel. The recipient *stream descriptor* can do whatever it likes with them. The default action is to ignore the kernel vectors.

2.3.2 Interface of a *CoeffSource*

To be written

2.3.3 Interface of an *SpTable*

To be written

3 The programs

In this section we describe the complementary program given with the library.

3.1 `analysis_fb`

3.2 `synthesis_fb`

3.3 `sptable_and_dual`

3.4 `recover_stream`

Appendix A Coefficient Stream Model

In this section we briefly describe the model we choose for the coefficient stream. The assumptions are weak enough to be adapted to any case of practical relevance.

Definitions:

Big and little time
: Every frame function (and every dual function) is labeled with a pair (c, k) where c, called *little time*, is a non-negative integer, and k, called *big time*, is a vector with integer entries. The little time can assume only a finite number of values. Let $N - 1$ be the maximum value c can assume.

  If the frame is obtained by means of oversampled filter banks, c represents the channel number, N is the number of channels and k represent the sampling istant.

  Note that this two-values labelling method can be made equivalent to a one-values one by letting $N = 1$.

Macro-coefficient
: The set of coefficients which share the same big time will be called a *macro-coefficient*. In the filter bank context, a macro-coefficient rapresent the vector of values which exit from the filter bank at a given time.

Closeness, neighborhood
: Two macro-coefficient with big times k and n will be said to be *separated* if every frame function with big time k is orthogonal to every dual function with big time n. Two macro-coefficient will be said *close* one another if they are not separated. It can be shown that closeness is a symmetric relation.

  The set of macro-coefficients which are close to a macro-coefficient C will be called the *neighborhood* of C.

Completeness
: If all the coefficients belonging to a macro-coefficient C were received, the macro-coefficient is said *complete*.

Processing time
: Macro-coefficients are sequentially read by `recover_stream()`. If macro-coefficient C is read at the n-th iteration, we will say that C has *processing time* equal to n. `recover_stream()` accesses the coefficient stream by asking for the macro-coefficient with a given processing time.

Waiting time
: The maximum among the processing times of the neighbors of C will be said the *waiting time* of C.

Appendix B File Formats

B.1 Format of Coefficient File

A coefficient file is composed of two part

1. A header line
2. A sequence of pairs (coefficients, known), one pair per line

B.1.1 The header line

The header line begins with the string `%FBstream%` followed by the following fields, separated by whitespaces

1. *major* and *minor* version number
2. Number of dimensions (D)
3. Number of channels (*n_channels*)
4. D integers representing the size of the coefficient set along each dimension
5. a case-insensitive string of D characters chosen among 'P', 'Z' and '0'. If the n-th character is 'P', then the signal was extended by periodicity along the n-th dimension, if the n-th character is 'Z' of '0', then the signal was padded with zeros along the n-th dimension,

Any character following the last field is ignored and it can be used to save remarks and other informations (for example, `analysis_fb` writes at the end of the header the filter bank used to produce the coefficient sequence).

Example

The header

```
%FBstream% 1 1 2 5 32 32 PP ../data/banks/daub4-2d.frame
```

is relative to a file with version 1.1 which contains a set of 32x32 coefficients obtained by applying a 5-channel filter bank (daub4-2d.frame) to a 2-dimensional signal. The signal was extended by periodicity along both dimensions.

B.1.2 Coefficient values

Each line after the header line has a two entries: a floating point number (in any format accepted by `C++`) and an integer number that can be only 0 or 1. If the second entry is 1, the coefficient has been received, if it is 0 the coefficient is lost. In the latter case the first field is never read and it can contain any value, but **it must be present**.

The coefficients are ordered as follows. Each coefficient is uniquely determined by D+1 values: the D coordinates plus the number of the channel. If we denote with $x(c; n_1, \ldots, n_D)$ the coefficient relative to channel `c` and coordinates $n_1, \ldots, n_D$, the coefficients are stored in the file by varying the first index more rapidaly, then the second and so on, similarly to the way used by Matlab® to transform a multidimensional array into a column vector.

Example

In the case of the header above, the coefficients would be stored on file according to the following order

```
x(1;  0,  0)
x(2;  0,  0)
x(3;  0,  0)
x(4;  0,  0)
x(5;  0,  0)
x(1;  1,  0)
x(2;  1,  0)
x(3;  1,  0)
x(4;  1,  0)
x(5;  1,  0)
x(1;  2,  0)
x(2;  2,  0)
       :
       :
x(5; 31,  0)
x(1;  0,  1)
       :
       :
x(5; 31, 31)
```

B.2 Signal Format

B.3 SP-Table Format

B.4 Filter Bank Format

Index

(Index is nonexistent)