The Libit library

Contents

Chapter 1

Introduction

LIBIT is a C library for information theory and signal processing. It extends the C language with vector, matrix, complex and function types, and provides some common source coding, channel coding and signal processing tools.

The goal of LIBIT is to provide easy to use yet efficient tools, and is mainly targeted at researchers and developpers in the fields of Communication and Compression. The syntax is purposedly close to that of other tools commonly used in these fields, such as MATLAB, OCTAVE, or IT++. Therefore, experiments and applications can be developped, ported and modified simply, without requiring deep knowledge of the C language. Additional goals of the library include portability to many platforms and architecture, and ease of installation.

Rather than trying to provide the very latest state-of-the-art techniques or a large panel of specific methods, this library aims at providing the most general and commonly used tools to build a communication chain, from signal processing and source coding to channel coding and transmission.

Among these tools are some common source and channel models, modulation and quantization techniques, wavelet analysis, entropy coding, etc... As examples and to ensure the correctness of the algorithms with respect to published results, some test programs are also provided.

All examples provided herein are small code snippets used to illustrate the document. They may not compile and run successfully.

As the library is still under heavy development, it is only partially documented currently.

Chapter 2

Basic Types

LIBIT defines some new basic types to extend the standard C types (char, int, double, float). They are used to handle some commonly used mathematical objects such as:

- complex numbers (cplx)
- vectors (Vec, vec, ivec, bvec, cvec)
- matrices (Mat, mat, imat, bmat, cmat)
- functions (it_function_t, it_ifunction_t)

2.1 Complex numbers

A new type cplx is used to represents complex numbers. For a given complex number c, the real part is accessed using `creal(c)`, while the imaginary part is accessed with `cimag(c)`. Both the real and imaginary parts are stored as double precision floating point numbers (double).

During declaration, a complex number can be initialized using the cplx macro. The first argument of the macro is the real part, the second is the imaginary part.

Basic operations are defined on complex numbers such as the addition (`cadd`), the subtraction (`csub`), the multiplication (`cmul`), and the division (`cdiv`). The inversion (`cinv`) can also be used instead of dividing one by a complex. The module of a complex is obtained using cnorm. To test a complex for equality with another complex, the `ceq` macro is defined, as the == operator is not available.

The operations available on complex numbers are summarized in the following table, where a,b are reals and x, y, z are complex numbers:

Operation	Expression
z = cplx(a,b)	z = $a + ib$
a = creal(x)	a = $\text{Re}(x)$
b = cimag(x)	b = $\text{Im}(x)$
z = cadd(x,y)	z = $x + y$
z = csub(x,y)	z = $x - y$
z = cmul(x,y)	z = $x * y$
z = cdiv(x,y)	z = x/y
z = cinv(x)	z = $1/x$
z = cconj(x)	z = x^*
z = cnorm(x)	z = $\|x\|$
ceq(x,y)	x == y

The following commonly used constant complex numbers are also defined:

Identifier	Value
cplx_0	0
cplx_1	1
cplx_I	i (=$\sqrt{-1}$)

The following example illustrates how to declare and use complex numbers:

Program 2.1: Complex number example

```
cplx x = cplx(1.5, 2.5), y = cplx(1.7, 2.1), z;

z = cmul(x, y); /* multiply x by y and store the result in z */
z = cconj(x);   /* conjugate of x */
```

2.2 Vectors

The vector type (Vec) allows to define vectors of elements of any type. These generic vectors are created with the `Vec_new` macro by specifying the type of element it contains and its initial length. For people familiar with C++, this is similar to a templated type. Vectors are used like C arrays, except their length is stored internally and can be changed dynamically. A vector length is accessed through the `Vec_length` macro and set using `Vec_set_length`. Vectors are destroyed with the `Vec_delete` call, releasing the allocated resources.

As vectors are actually pointers to their elements, they can be used in any place where a pointer type to the element is needed. The element size (in bytes) of a vector is also accessible using the

Vector type	Element type
vec	double
ivec	int
bvec	unsigned char
cvec	cplx

Table 2.1: Vectors types

`Vec_element_size` macro. This allows for a great flexibility due to compatibility with existing C functions. For example to read the content of a vector from a binary file, the following code can be used:

Program 2.2: Vector example

```
Vec v = Vec_new(float, 10); /* create a new vector of 10 float elements */
/* fill the vector with 10 floats read from a binary file identified by 'fd' */
fread(v, Vec_element_size(v), Vec_length(v), fd);
Vec_delete(v);              /* release the resources used by v */
```

For practical reasons and ease of use, this generic Vec type is derived into four commonly used vector types depicted Table 2.2:

Vector elements are accessed with the bracket operator [] starting at index 0. For instance, `v[3]` corresponds to the fourth element of vector `v`. Functions requiring an index can be given the 'end' keyword instead, which corresponds to the last index of the vector to process.

Each vector type has specific functions to handle it, which perform some additional type checking. Therefore, although the length of a ivec can be retrieved using the `Vec_length` macro, the `ivec_length` function is prefered. These specific functions being defined for all types (i.e. `vec_length`, `ivec_length`, `bvec_length`, `cvec_length`), only the `vec_` variants will be documented here when there is no ambiguity on how to derive the `ivec_`, `bvec_` and `cvec_` variants.

Vectors can be copied using the `vec_clone()` function. This will create a new copy of the vector, properly allocated and with each element copied independently. Using the equal sign (=) will *not* copy a vector, rather create a reference to it (modifying one will modify the other). Similarly, testing vectors for equality is done using the `vec_eq()` function instead of the == operator. Also, many functions, such as `vec_set_length()` which sets the length of a vector, may modify the actual value of the vector pointer. References created using the equal sign are then *invalid*. Unless you know what you're doing, it is generally better never to use the equal sign on a vector and prefer the `vec_clone()` call instead.

Program 2.3: Copy, reference, and test for equality

```
vec v = vec_new(2);   /* create a new vector of 2 double elements */
v[0] = 0.0;
v[1] = 0.5;
vec v1 = v;            /* create a reference to the vector v */
vec v2 = vec_clone(v); /* create a copy of the vector v */
if(v2 == v)            /* v2 == v ? false */
  printf("same object\n");
if(vec_eq(v2,v))       /* content of v2 same as v ? true */
  printf("same vector\n");
v1[0] = 1.0;
v2[1] = 2.0;
/* v contains [ 1.0 0.5 ] */
/* v1 contains [ 1.0 0.5 ] (always the same as v) */
/* v2 contains [ 0.0 2.0 ] */
```

Many common vectors are created easily. Constant vectors, containing the same element repeated over the length of the vector, are created using the `vec_new_set()` call. In particular a vector full of zero or full of one is created using the `vec_new_zeros()` or `vec_new_ones()` call respectively. Arithmetic series are created using the `vec_new_arithm()` call. In particular the vector $0, \ldots, N-1$ is created with `vec_new_range(N)` and the vector $1, \ldots, N$ is created using `vec_new_1N(N)`. Similarly, the `vec_new_geom()` function allows to create geometric series. One particular case is the complex vectors of the roots of unity, created using `cvec_new_unit_roots()`.

Vectors can also be created directly from a string using `vec_new_string()` by specifying the value of each element. For more information on parsing and initialization from strings see Chapter 3.

All vector creation functions exist in a direct form allowing their use on an already created vector. They generally have the same function names, except for the '_new'. For example `vec_set()` sets the value of an existing vector to a constant value.

Program 2.4: Common vectors

```
vec v0 = vec_new_zeros(5);    /* create the vector [ 0 0 0 0 0 ]   */
vec v1 = vec_new_ones(5);     /* create the vector [ 1 1 1 1 1 ]   */
vec v2 = vec_new_set(2.3, 3); /* create the vector [ 2.3 2.3 2.3 ] */
vec v3 = vec_new_1N(5);       /* create the vector [ 1 2 3 4 5 ]   */
vec v4 = vec_new_range(5);    /* create the vector [ 0 1 2 3 4 ]   */
vec v5 = vec_new_arithm(1.5, 0.1, 3); /* vector [ 1.5 1.6 1.7 ]    */
vec v6 = vec_new_geom(1.5, 0.1, 3);   /* vector [ 1.5 0.15 0.015 ] */
cvec v7 = cvec_new_unit_roots(4);     /* vector [ 1 i -1 -i ]      */
vec v8 = vec_new_string("0.1 0.3 0.4"); /* vector [ 0.1 0.3 0.4 ]  */
```

```
vec_ones(v3);                          /* set v3 to [ 1 1 1 1 1 ] */
```

The most common arithmetic operations are defined on vectors. Scalar operations include `vec_incr()`, `vec_decr()`, `vec_mul_by()` and `vec_div_by()`, which respectively add, subtract, multiply or divide each element of a vector with a scalar element. Elementwise operations between vectors are performed using the `vec_add()`, `vec_sub()`, `vec_mul()`, `vec_div()` functions which respectively add, subtract, multiply or divide each element of a first vector with the corresponding element in the second vector. The inner product between two vectors is computed with `vec_inner_product()`. The concatenation of two vectors is performed using the `vec_concat()` function.

Mathematical functions can be applied to vectors using the `vec_apply_function()` call. Functions are defined using the it_function_t type, for more information see Section 2.4. Commonly used functions, such as the exponential, natural logarithm, base-10 logarithm, negation, square, absolute value, and power are applied using the `vec_exp()`, `vec_log()`, `vec_log10()`, `vec_neg()`, `vec_sqr()`, `vec_abs()` or `vec_pow()` functions respectively.

The sum of a vector is computed using `vec_sum()`. It can be computed partially between two indexes using the `vec_sum_between()` call. Vectors are normalized using `vec_normalize()`, resulting in a vector whose sum is equal to one. The mean of a vector is obtained by `vec_mean()`, while the median is obtained using `vec_median()`. The unbiased variance of a vector can be computed with the `vec_variance()` function, whereas the norm of a vector is computed with `vec_norm()`.

Program 2.5: Operations on vectors

```
vec v1 = vec_new_string("0.3 0.5 1");  /* vector [ 0.3 0.5 1 ]     */
vec v2 = vec_new_string("2 -0.7 1.5"); /* vector [ 2 -0.7 1.5 ]    */
vec_incr(v1, 1.3);                     /* v1 = [ 1.6 1.8 2.3 ]     */
vec_mul_by(v1, 2);                     /* v1 = [ 3.2 3.6 4.6 ]     */
vec_add(v1, v2);                       /* v1 = [ 5.2 2.9 2.5 ]     */
double p = vec_inner_product(v1, v2);  /* p = 12.12                */
vec_range(v1);                         /* v1 = [ 1 2 3 ]           */
vec_exp(v1);                           /* v1 = [ 2.72 7.39 20.09 ] */
vec_log(v1);                           /* v1 = [ 1 2 3 ]           */
vec_neg(v1);                           /* v1 = [ -1 -2 -3 ]        */
vec_sqr(v1);                           /* v1 = [ 1 4 9 ]           */
double s = vec_sum(v1);                /* s = 1+4+9 = 14           */
s = vec_sum_between(v1, 1, end);       /* s =   4+9 = 13           */
double mean = vec_mean(v1);            /* mean = 14/3 = 4.67       */
double var = vec_variance(v1);         /* var = 16.33              */
```

Minimum and maximum values of a vector are obtained with the `vec_min()` and `vec_max()` calls respectively. The index in the vector where the minimum or maximum occurs (argmin, argmax) are obtained with the `vec_min_index()` and `vec_max_index()` functions.

Vectors are sorted in increasing order using the `vec_sort()` or `ivec_sort()` call. A vector containing the indexes corresponding to increasing values of the input vector can be created using the `vec_sort_index()` and `ivec_sort_index()` function. Since complex numbers are unordered, these functions are not defined for cvec. The current algorithm for sorting is the quick sort. Vectors can be reversed using `vec_reverse()` function, the first element being exchanged with the last element and so forth. The number of occurences of a value in a vector is available through the `vec_count()` function. Searching is performed using `vec_find` or `vec_find_first()` which respectively return a vector of indexes of the search value or the index of the first occurence of the value. Similarly, `vec_replace()` replaces each occurences of a given value with another, returning the vector of indexes of the replaced positions. The `vec_index_by()` functions allows to shuffle a vector by creating a new vectors composed of elements of the input vector indexed by the index vector. Note that indexes can appear more than once in the indexing vector. This function is particularly useful for interleaving or dequantizing.

Program 2.6: Sorting vectors

```
vec v = vec_new_string("2 -0.7 1.5");  /* vector [ 2 -0.7 1.5 ]         */
double min = vec_min(v);               /* min = -0.7                    */
int idx = vec_min_index(v);            /* idx = 1                       */
ivec iv = vec_qsort_index(v);          /* iv = [ 1 2 0 ]                */
vec_qsort(v);                          /* v = [ -0.7 1.5 2 ]            */
vec_reverse(v);                        /* v = [ 2 1.5 -0.7 ]            */
iv = vec_replace(v, 1.5, -1);          /* v = [ 2 -1 -0.7 ]; iv = [ 1 ] */
iv = ivec_new_string("1 1 2");         /* iv = [ 1 1 2 ] */
v = vec_index_by(v, iv);               /* v = [ -1 -1 -0.7 ] */
```

Stacks are efficiently implemented using vectors. A new element is added on top the stack using the `vec_push()` call and removed using the `vec_pop()` call. The topmost element is accessed using `vec_head()`. Due to the use of geometric reallocation, this operation is implemented in $O(N)$ time, where N is the size of the vector. Similarly, vectors can be used as lists using the `vec_ins()` call to insert an element and `vec_del()` call to delete an element. Insertion and deletion are in $O(N)$ time, with access in $O(1)$ time. Finally, vectors can be used to represent sets, where a value is appearing only once in the vector. The `vec_unique()` call creates a vector with exactly one element for each value present in the input vector. The `vec_union()` and `vec_intersection()` call create a vector where each element is a value appearing exactly once in either or both vectors respectively.

Program 2.7: Stacks, lists and sets

Matrix type	Element type
mat	double
imat	int
bmat	unsigned char
cmat	cplx

Table 2.2: Matrix types

```
vec v = vec_new(0);             /* empty vector */
vec_push(v, 1.0);               /* v = [ 1.0 ]               */
vec_push(v, 1.1);               /* v = [ 1.0 1.1 ]           */
vec_push(v, 3.0);               /* v = [ 1.0 1.1 3.0 ]       */
double h = vec_head(v);         /* h = 3.0                   */
vec_pop(v);                     /* v = [ 1.0 1.1 ]           */
vec_ins(v, 1, 2.0);             /* v = [ 1.0 2.0 1.1 ]       */
vec_ins(v, 3, 2.0);             /* v = [ 1.0 2.0 1.1 2.0 ] */
vec_del(v, 2);                  /* v = [ 1.0 2.0 2.0 ]       */
vec s = vec_unique(v);          /* s = [ 1.0 2.0 ]           */
v = vec_new_string("0.8 1.0");  /* v = [ 0.8 1.0 ]           */
vec u = vec_union(s, v);        /* u = [ 0.8 1.0 2.0 ]       */
vec i = vec_intersection(s, v); /* i = [ 1.0 ]               */
```

Vectors are displayed using the `it_printf()` family of functions. For more information on how to display the new types see Chapter 3.

Program 2.8: Printing

```
cvec v = cvec_new_unit_roots(4);      /* vector [ 1 i -1 -i ]      */
it_printf("$z\n", v); /* [1.000000 +1.000000i -1.000000 -1.000000i] */
```

2.3 Matrices

Matrices are build upon vectors in a very similar manner. A generic Mat type is provided to allow the definition of any type of matrix. These generic matrices are created using the `Mat_new()` macro by specifying the type of the elements along with the width and height of the matrix. They are deleted using `Mat_delete()` and their width and height can be get or set using `Mat_width()`, `Mat_height()`, `Mat_set_width()` and `Mat_set_height()` respectively. Similarly to vectors, the generic Mat type is derived into four types for the most common usage:

Matrix elements are accessed with the double bracket operator [][], which starts at index 0 in the (row,column) order. For example, $m[2][3]$ corresponds to the element at the third row and fourth column of the matrix m. Each row of the matrix is a vector which can be accessed using the bracket operator []. Thus, $m[2]$ is the vector corresponding to the third row of the matrix. As for vectors, functions requiring row or column indexes can be given the 'end' keyword corresponding to the index of the last row or last column respectively. As many functions are defined similarly on the mat, imat, bmat and cmat types, only the mat functions will be documented here when there is no ambiguity on how to derive the corresponding functions for the other matrix types.

Matrices are copied using the `mat_clone()` function. Using the equal sign (=) will *not* copy a matrix, rather create a reference to it (modifying one will modify the other). Similarly, testing matrices for equality is done using the `mat_eq()` function instead of the == operator. Also, many functions, such as `mat_set_height()` which sets the height of a matrix, may modify the actual value of the matrix pointer. References created using the equal sign are then *invalid*. Unless you know what you're doing, it is generally better never to use the equal sign on a matrix and prefer the `mat_clone()` call instead.

Program 2.9: Copy, reference, and test for equality

```
mat m = mat_new(2,2);  /* create a new 2x2 matrix of double elements */
m[0][0] = 0.0;
m[0][1] = 0.5;         /* set 2nd element of the 1st row to 0.5 */
vec_set(m[1], 1.4);    /* set the whole 2nd row to 1.4 */
mat m1 = m;            /* create a reference to the matrix m */
mat m2 = mat_clone(m); /* create a copy of the matrix m */
if(m2 == m)            /* m2 == m ? false */
  printf("same object\n");
if(mat_eq(m2,m))       /* content of m2 same as m ? true */
  printf("same matrix\n");
m1[0][1] = 1.0;
m2[1][0] = 2.0;
/* m contains [[ 0.0 1.0 ] [ 1.4 1.4 ]] */
/* m1 contains [[ 0.0 1.0 ] [ 1.4 1.4 ]] */
/* m2 contains [[ 0.0 0.5 ] [ 2.0 1.4 ]] */
```

Constant matrices, containing the same element repeated over the length of the vector, are created using the `mat_new_set()` call. In particular a matrix full of zero or full of one is created using the `mat_new_zeros()` or `mat_new_ones()` call respectively. A diagonal matrix is created from a vector using `mat_new_diag()`. The identity matrix is created using `mat_new_eye()`. All these functions also exist in a direct form, to be used on an already existing matrix (`mat_set`, `mat_zeros`, `mat_ones`, `mat_diag`, `mat_eye`).

Program 2.10: Common matrices

```
mat m0 = mat_new_zeros(1, 2);     /* create the matrix [[0 0]]       */
mat m1 = mat_new_ones(2, 1);      /* create the matrix [[1] [1]]     */
mat m2 = mat_new_set(3, 2, 2);    /* create the matrix [[3 3] [3 3]] */
cvec cv = cvec_new_unit_roots(4); /* vector [ 1 i -1 -i ]            */
cmat m3 = cmat_new_diag(cv);      /* create the matrix [[ 1  0  0  0 ]
                                                        [ 0  i  0  0 ]
                                                        [ 0  0 -1  0 ]
                                                        [ 0  0  0 -i ]] */
mat m4 = mat_new_eye(2);          /* create the matrix [[1 0] [0 1]] */

mat_ones(m4);                     /* set m4 to [[1 1] [1 1]]         */
```

Submatrices can be extracted from a matrix using the function `mat_get_submatrix()`. A rectangle in a given matrix can be set using `mat_set_submatrix()`. Similarly, `mat_get_col()`, `mat_get_row()`, `mat_set_col()` and `mat_set_row()` allow the retrieval or the setting a row or a column of a matrix. Matrices can be turned into vectors using `mat_to_vec()` which stores the element of a matrix in a vector in raster order (fill a row with elements of the vector, then proceed to next row). Given the width of the matrix, a vector can also be turned into matrix using the `vec_to_mat()` call. Note that `mat_get_submatrix()`, `mat_get_row()`, `mat_get_col()`, `mat_to_vec()` and `vec_to_mat()` are constructive functions, allocating a new matrix or vector object each time they are called.

Program 2.11: Submatrices

```
cvec cv = cvec_new_unit_roots(4); /* vector [ 1 i -1 -i ]               */
cmat m1 = cmat_new_diag(cv);      /* create the matrix [[ 1  0  0  0 ]
                                                        [ 0  i  0  0 ]
                                                        [ 0  0 -1  0 ]
                                                        [ 0  0  0 -i ]] */
cmat m2 = cmat_get_submatrix(m1, 2, 2, end, end);
                                  /* create the matrix [[-1 0] [0 -i]]  */
cvec_ones(m2);                    /* m2 = [[1 1] [1 1]]                 */
cmat_set_submatrix(m1, m2, 1, 1); /* m1 = [[ 1  0  0  0 ]
                                           [ 0  1  1  0 ]
                                           [ 0  1  1  0 ]
                                           [ 0  0  0 -i ]]              */
cvec cv2 = cmat_get_col(m1, 3);   /* v = [ 0 0 0 -i ]                   */
cmat_set_row(m1, 1, cv);          /* m1 = [[ 1  0  0  0 ]
                                           [ 1  i -1 -i ]
                                           [ 0  1  1  0 ]
```

```
                                          [ 0  0  0 -i ]]           */
cvec cv3 = cmat_to_cvec(m1);
                        /* cv3 = [ 1 0 0 0 1 i -1 -i 0 1 1 0 0 0 0 -i ] */
m3 = cvec_to_cmat(cv3, 8);        /* m3 = [[ 1  0  0  0  1  i -1 -i ]
                                          [ 0  1  1  0  0  0  0 -i ]]  */
```

Some common arithmetic operations are defined on matrices. Scalar operations include `mat_incr()`, `mat_decr()`, `mat_mul_by()` and `mat_div_by()`, which respectively add, subtract, multiply or divide each element of a matrix with a scalar element. These operations may also be performed only on a specific row or column of the matrix by adding '_row' or '_col' to the function name (e.g. `mat_col_incr()`). Elementwise operations between matrices are performed using the `mat_elem_add()`, `mat_elem_sub()`, `mat_elem_mul()`, `mat_elem_div()` functions which respectively add, subtract, multiply or divide each element of a first matrix with the corresponding element in the matrix vector. Matrices are added or subtracted using `mat_add()` or `mat_sub()` respectively (or equivalently `vec_elem_add` and `vec_elem_sub`). The product of two matrices, of a matrix with a vector, or of a vector with a matrix is obtained using `mat_mul()`, `mat_vec_mul()` or `vec_mat_mul()` respectively. To transpose a matrix, use `mat_transpose()`. Matrices are inverted using `mat_inv()`. Note that constructive versions of these last two functions are defined (`mat_new_transpose()`, `mat_new_inv()`).

Mathematical functions can be applied to matrices using the `mat_apply_function()` call. Functions are defined using the it_function_t type, for more information see 2.4. Minimum and maximum values of a matrix are obtained with the `mat_min()` and `mat_max()` calls respectively.

The sum of all elements in a matrix is computed using `mat_sum()`. It can also be computed only on a specific row or column using `mat_row_sum()` or `mat_col_sum()` respectively. Additionnally, the vector corresponding to the sum of each columns or the sum of each rows is available through `mat_cols_sum()` or `mat_rows_sum()` respectively.

Matrices are normalized using `mat_normalize()`, resulting in a matrix whose sum is equal to one. The mean of a matrix is obtained using `mat_mean()`. The unbiased variance of a matrix can be computed with the `mat_variance()` function. The one and infinite norms are obtained using mat_norm_1() and `mat_norm_inf()` respectively.

Program 2.12: Operations on matrices

```
mat m1 = mat_new_set(4, 2, 2);          /* matrix [[4 4] [4 4]]      */
mat m2 = mat_new_eye(2);                /* matrix [[1 0] [0 1]]      */
mat_incr(m1, 2);                        /* m1 = [[6 6] [6 6]]        */
mat_col_incr(m1, 2, 4);                 /* m1 = [[6 10] [6 10]]      */
mat_div_by(m1, 2);                      /* m1 = [[3 5] [3 5]]        */
mat_add(m1, m2);                        /* m1 = [[4 5] [3 6]]        */
mat_mul(m1, m2);                        /* m1 = [[4 5] [3 6]]        */
```

```
mat_transpose(m1, m2);                  /* m1 = [[4 3] [5 6]]       */
mat_elem_mul(m1, m2);                   /* m1 = [[4 0] [0 6]]       */

mat_eval(m1, IT_FUNCTION(sqrt), NULL); /* m1 = [[2 0] [0 2.45 ]]   */
m1[1][0] = 1;                           /* m1 = [[2 0] [1 2.45 ]]   */
double s = mat_sum(m1);                 /* s = 5.45                 */
vec v = mat_cols_sum(m1);               /* v = [ [3] [2.45] ]       */
double mean = mat_mean(m1);             /* mean = 5.45/4 = 1.36     */
double norm1 = mat_norm_1(m1);          /* 1-norm = 3.45            */
double normI = mat_norm_inf(m1);        /* infinite-norm = 3        */
```

Matrices are displayed using the `it_printf()` family of functions. For more information on how to display the new types see 3.

Program 2.13: Printing

```
mat v = mat_new_eye(4);       /* identity matrix in dim. 4 */
it_printf("#f\n", v); /* [[1.000000 0.000000 0.000000 0.000000]  */
                        [0.000000 1.000000 0.000000 0.000000]  */
                        [0.000000 0.000000 1.000000 0.000000]  */
                        [0.000000 0.000000 0.000000 1.000000]] */
```

2.4 Functions

In order to handle continuous functions (e.g. for probability density functions) a new it_function_t type is defined. Functions have only one variable 'x' of double type, and return a single real value of double type too. They are declared using the `it_function()` macro, with no return value and no argument. Here is a simple example of a function called 'normal' returning the value of a gaussian pdf with zero mean and variance equal to one.

Program 2.14: Function example

```
it\_function(normal)
{
  return(1/sqrt(2.0*M_PI) * exp(-x*x / 2.0));
}
```

Although functions are always univariate, they can have extra parameters to modify they behaviour. Function parameters are declared using the `it_function_args()` macro. Each parameter is declared in a struct-like manner. A function parameters is accessed as an element of the it_this structure. For example we can now define a function called 'gaussian' with a varying parameter sigma representing the standard deviation of the gaussian:

Program 2.15: Function example (with parameters)

```
/* the gaussian function with parameter sigma */
it_function_args(gaussian) {
  double sigma;  /* standard deviation */
};
it_function(gaussian)
{
  double sigma = it_this->sigma;

  return(1.0 / (sqrt(2.0*M_PI)*sigma) * exp(-x*x / (2.0*sigma*sigma)));
}
```

There are some predefined functions in LIBIT. One is 'itf_identity', the identity function (which returns x). Other are operators which allow to perform basic operations on functions, such as addition, multiplication, composition, differentiation and integration. An operator is actually a function taking one or more function and its arguments as parameters. Here is an example on how to build a function which is the product of the identity and our previously defined gaussian:

Program 2.16: Multiplication operator example

```
/* declare the gaussian parameters */
it_function_args(gaussian) gaussian_args;
/* declare the multiplication operator parameters */
it_function_args(mul) mul_args;

/* function product */
mul_args.f = itf_identity; /* first operand: the identity function  */
mul_args.f_args = NULL;    /* which takes no special parameters.    */
mul_args.g = gaussian;     /* second operand: our gaussian function */
mul_args.g_args = &gaussian_args;  /* and its arguments (sigma) */

gaussian_args.sigma = 2;   /* set sigma to 2 */
itf_mul(2, &mul_args); /* compute x*g(x, sigma) for x = 2, sigma = 2 */
itf_mul(1, &mul_args); /* compute x*g(x, sigma) for x = 1, sigma = 2 */
gaussian_args.sigma = 3;   /* set sigma to 3 */
itf_mul(2, &mul_args); /* compute x*g(x, sigma) for x = 2, sigma = 3 */
```

Since operators are just another kind of function, they can be composed easily. Standard C functions taking only a double as their argument (like many functions of math.h) can be cast into it_function_t using the `IT_FUNCTION()` macro. Here is another example on how to compute the derivative of the arctangent:

Program 2.17: Differentiation example

```
/* derivatives */
differentiate_args.function = IT_FUNCTION(atan); /* the arctangent function */
differentiate_args.args = NULL;
itf_differentiate(2.0, &differentiate_args); /* compute the derivative of
                                                 the arctangent in 2 (=1/5) */
```

Note that there exists also a similar it_ifunction_t type for functions of a unique integer variable returning an integer variable. Multivariate functions taking a input vector and returning a double are also defined as it_vfunction, however no operation is defined on these objects yet.

Chapter 3

Input/Output

3.1 Printing vector, matrices and complex numbers

LIBIT provides a new set of printf functions to handle the vector and matrix types easily. Since the `it_printf` function is based on the C `printf` function, it supports all the basic types (%c,%d,%s,...) and modifiers (%.03f). The `it_fprintf` and `it_vfprintf` functions are also provided as extensions to the fprintf and vfprintf functions.

An additional complex type (cplx) is defined in LIBIT and can be printed using letter z. Complex numbers are printed as a pair of double floats in the form 'a + b * i' and therefore accept the same modifiers as double values.

Vectors of any type can be printed by substituting the character '$' to the character '%' in the format string. For example a vector of complex (cvec) is printed by using the format string "$z". Vectors are printed as a sequence of elements, enclosed in brackets. Modifiers in the vector format string are applied to all printed elements of the vector. For example "$.3f" prints a vector of floats all limited to 3 decimals. Additionally, there is a default type 'v' to print floats similarly to MATLAB (tm). The default is to limit floats to 3 decimals. It can be changed using the `it_set_vec_default_fmt()` call. In the default setting, "$v" is therefore equivalent to "$.3f".

Similarly, matrices are printed by substituting the character '#' to the character '%' in the format string. Therefore, a matrix of double (mat), is printed using the "#f" format string. Matrices are printed with each row on a line, printed as a vector (with brackets), with all lines enclosed in two additional brackets. All modifiers apply to the matrix elements as in vectors. Besides, a default format 'm' is defined. It is equivaled to "9.3f" in the default setting and can be changed with the `it_set_mat_default_fmt()` call.

In order to print the character '#' or '$', the format string "##" or "$$" can be used respectively.

Here is an example of the various extensions to the printf call.

Program 3.18: Printing example

```
vec v = vec_new_ones(5);     /* create a new vector of 5 elements set to 1 */
it_printf("$v\n", v);        /* print this vector as
                                  [ 1.000 1.000 1.000 1.000 1.000 ]        */

mat m = mat_new_zeros(2, 3); /* create a new matrix of 2 rows and 3 columns */
it_printf("#m\n", m);        /* print the matrix as
                                 [[    0.000    0.000    0.000]
                                  [    0.000    0.000    0.000]]            */

cplx c = cplx(2, 3);         /* create the complex number 2 + 3i */
it_printf("%z\n", c);        /* print the complex as
                                 2.000000 + 3.000000 * i                    */
```

3.2 Image reading and writing

Images can be read in the PGM format as matrices using the `mat_pgm_read()` or `imat_pgm_read()` functions. Both these functions return the matrix of pixels, representing intensities between 0 and the maximum value specified in the PGM (generally 255). Similarly, images can be written to disk using `mat_pgm_write()` or `imat_pgm_write()`.The sample values are read as integers and written with proper rounding and clipping. Some information can be retrieved from any PNM file using `pnm_info()`, for example to check the image is in a supported format (PGM type 5 currently).

Program 3.19: Image input/output

```
imat m;

m = imat_pgm_read("image.pgm");  /* read the image      */
mat_transpose(m);                /* transpose the image */
imat_pgm_write(m, "image.pgm");  /* write the image     */

char comments[1000]; /* the comments in the PNM       */
int width, height;   /* width and height of the image */
int pnm_type;        /* type of the PNM file          */
int maxval;          /* maximum intensity value       */
pnm_info("image.pgm",  &pnm_type,  &width,  &height, &maxval, comments, 1000);
```

3.3 Sound reading and writing

Sound files can be read in the WAV format as matrices by the following calls: `mat_wav_read()` and `imat_wav_read()`. A row of the matrix corresponds to a channel (i.e. stereo files are read as a two-row matrix). The columns correspond to the instants in time. Similarly, sounds can be written to disk using `mat_wav_write()` or `imat_wav_write()` call, provided the sampling rate and sample resolution is also given. Some information can be retrieved from any WAV file using `wav_info()` call. Compressed WAV files are not supported.

Program 3.20: Sound input/output

```
imat m;

m = imat_wav_read("sound.wav"); /* read the sound            */
vec_reverse(m[0]);              /* reverse the first channel */
imat_wav_write(m, "sound.wav"); /* write the sound           */

int channels;         /* number of channels         */
int srate;            /* sampling rate              */
int depth;            /* sampling resolution in bits */
int length;           /* number of samples          */
wav_info( sound_in, &channels, &srate, &depth, &length );
```

3.4 Parser

In order to manage sources of external data, a parser is provided. The parser allows to handle three kind of sources:

- command line arguments,
- parameter files,
- strings.

These sources are fed to the initialization function `parser_init` by a call of the form

```
parser = parser_init(argc, argv, "param.dat", "a=-35\nb=3\n");
```

Note that if a variable is defined in two sources handled by the same parser, the data will be retrieved in the order of the initialization function argument. Hence, the latter kind of source (strings) allows one to easily manage the default arguments. If extra sources are added to the

Kind of Source	Extra source Function
Command line arguments	parser_add_params
File	parser_add_file
String	parser_add_string

Table 3.1: Parser source adding functions

Function name	variable type
parser_get_int	int
parser_get_double	double
parser_get_byte	byte
parser_get_cplx	cplx
parser_get_string	char *
parser_get_vec	vec
parser_get_ivec	ivec
parser_get_bvec	bvec
parser_get_cvec	cvec
parser_get_mat	mat
parser_get_imat	imat
parser_get_bmat	bmat

Table 3.2: Functions to retrieve variables with a parser

parser (see below), the priority is given by the order in which the sources have been added to the parser.

There is also a function per kind of source that adds the source to the parser internal data. Note that, to use one of these functions the parser must previously been allocated by an initialization function. The table below summarizes the list of functions for each kind of source.

The three kinds of source may be useful in the same program. Hence, a source added while initializing the parser is always with the highest priority. The variables are retrieved with the functions summarized in the table below. Note that the objects are initialized by the parser and must be freed afterwards.

The parser type is `parser_t`. As in most of advanced types, the user should not handle directly `parser_t` object, but a pointer `parser_t *`. The corresponding memory area is allocated with the functions `parser_init_params`, `parser_init_file` or `parser_init_string`. For a typical program, the following order of priority between sources is of interest: command line arguments, parameter file, string for default parameters. The program below illustrates this scenario.

Program 3.21: Parser example

```
/* Initialization of the parser with all sources                  */
parser_t * parser = parser_init( argc, argv, "param.dat", "a=-35" ); */

/* Add some default values if the parameters have not been defined */
parser_add_string( parser, "def=9\nmyvec=[-1,-2,-3]" );

/* Print the content pre-processed by the parser                  */
parser_print( parser );

/* Retrieve some variables with the parser                        */
i   = parser_get_int(parser, "a");
def = parser_get_int(parser, "def");
dbl = parser_get_double(parser, "dbl");
s   = parser_get_string(parser, "S");
v   = parser_get_vec(parser, "myvec");       /* By default [1 2 3] */
iv  = parser_get_ivec(parser, "myivec");

/* Test if the variable myivec is defined                         */
if( !parser_exists(parser, "myivec") )
  it_warning("Variable myivec is not defined\n");

/* Free the pointer                                               */
parser_delete(parser);
free(s);
vec_delete(v);                             /* Vectors must be free */
ivec_delete(v);
```

Chapter 4

Measures

This chapter presents some various measures implanted in LIBIT, including the ones encountered in coding and information theory.

4.1 Distance measures

Various distances are provided to measure differences between vectors, such as the Hamming distance and the symbol or bit error rate (SER/BER), the Levenshtein distance, and the norm distance and mean square error (MSE).

The Hamming distance is the number of elements that are not equal in the two vectors. If the vectors are not of the same size, the distance is increased by the difference of lengths (i.e. the missing symbols are assumed to be not equal). Divided by the length of the original vector, this leads to the symbol error rate (SER). The Hamming distance is computed using `vec_distance_hamming()`. The SER is computed using `vec_ser()`, `ivec_ser()` and `bvec_ber()`. If the 'received' vector (second parameter) has a larger size as the 'original' vector (first parameter), the excess symbols are discarded for computing the SER.

The Levenshtein distance corresponds to the minimum of insertion, deletion or substitutions needed to transform one vector into another. The costs need not be equal for all operations although they are often assumed so. The function `ivec_distance_levenshtein()` is provided to compute this distance.

The Euclidean norm between two vectors, corresponding the norm of the difference of the vectors, is computed using `vec_distance_norm()`. It is defined the same way for matrices (e.g. to compute the norm between images) using `mat_distance_norm()`. The norm to use is given as the power factor which is often assigned the value 2. If the vectors are not of the same size, the distance is increased assuming that missing elements are equal to 0. The mean square error is computed by dividing the 2-norm between two vectors by the number of elements, and obtained from `vec_distance_mse()` and `mat_distance_mse()`. Missing elements are assumed to be equal to a parameter specifying the default reconstruction value.

Program 4.22: Distance example

```
ivec v1 = ivec_new_string("1 2 3");           /* v1 = [ 1 2 3 ]        */
ivec v2 = ivec_new_string("0 2 5");           /* v2 = [ 0 2 5 ]        */
ivec v3 = ivec_new_string("0 1 2 3");         /* v3 = [ 0 2 2 3 ]      */

/* compute the hamming distance between v1 and v2 */
ivec_distance_hamming(v1, v2);                /* hamming distance = 2  */

/* compute the SER between v1 and v2 */
ivec_ser(v1, v2);                             /* SER = 2/3 = 0.667     */

/* compute the Levenshtein distance between v1 and v3 */
ivec_distance_levenshtein(v1, v3, 1, 1, 1);   /* deletion+subst. => 2  */

/* compute the MSE between v1 and v2 */
ivec_distance_mse(v1,v2,0);                   /* (1 + 2^2) / 3 = 1.667 */
```

4.2 PDF estimation and measure

The PDF (probability density function) of a stationary ergodic source can be estimated from the histogram of the samples of that source. The histogram of a vector of samples is obtained using `histogram()`, returning the number of occurences of each particular value of the alphabet in the input vector. The normalized histogram, representing an estimator of the PDF, is obtained using `histogram_normalized()`. The cardinal of the set of integer values taken by the source must be given to these functions, and the input vector is assumed to be positive. The conditional histogram, that is the bidimensional histogram of a sample knowing the previous sample, is computed similarly using `histogram_cond()`.

Program 4.23: Histogram example

```
ivec v = ivec_new_string("0 0 1 0 1 1 1 0 0 0"); /* [ 0 0 1 0 1 1 1 0 0 0 ] */

/* compute the histogram of v assumed to be a binary source */
ivec h = histogram(2, v);                       /* h = [ 6 4 ]             */

/* estimate the first-order PDF of v */
vec pdf = histogram_normalized(2, v);           /* pdf = [ 0.6 0.4 ]       */
```

```
/* compute the conditional histogram of v */
imat cpdf = histogram_cond(2, v);                /* cpdf = [ [ 3 2 ]
                                                             [ 2 2 ] ]      */
```

To compute the stationary PDF of a Markov chain from the transition probability matrix the function `markov_marg_pdf()` is provided. It uses the Froebenius theorem which states that the stationary PDF is obtained from the eigen vector of the transition probability matrix associated with the eigenvalue 1.

Program 4.24: Stationary law estimation example

```
ivec v = ivec_new_string("0 0 1 0 1 1 1 0 0 0"); /* [ 0 0 1 0 1 1 1 0 0 0 ]  */

/* compute the conditional PDF of v */
imat ch = histogram_cond(2, v);                 /* ch = [ [ 3 2 ]
                                                           [ 2 2 ] ]         */
mat cpdf = mat_new(2, 2);
int s = imat_sum_col(cpdf, 0);
cpdf[0][0] = (double) ch[0][0] / s;
cpdf[1][0] = (double) ch[1][0] / s;
int s = imat_sum_col(cpdf, 1);
cpdf[0][1] = (double) ch[0][1] / s;
cpdf[1][1] = (double) ch[1][1] / s;
                                                /* cpdf = [ [ 0.6 0.5 ]
                                                             [ 0.4 0.5 ] ]   */

/* estimate the stationary law */
vec pdf = markov_marg_pdf(cpdf);                /* pdf = [0.555 0.444]      */
                                                /*     = [5/9 4/9]          */
```

The expectation (first-order moment) and variance (second-order moment) of a discrete source are computed from the PDF and the values associated to each symbol using `source_expectation()` and `source_variance()` respectively.

Program 4.25: Expectation and variance example

```
vec pdf = vec_new_string("0.5 0.3 0.1 0.1"); /* probability density function */
vec rec = vec_new_string("-1 0 1.5 2.5");    /* values of the symbols        */

/* compute the expectation of the source */
```

```
double mean = source_expectation(pdf, rec);  /* -0.5 + 0.15 + 0.25 = -0.1    */

/* compute the variance of the source */
double var = source_variance(pdf, rec);      /* 0.5 + 0.225 + 0.625 = 1.35   */
```

A PDF can be check for validity using `is_valid_pdf()`, by verifying it sums to one up to a small rounding error. Similarly a transition probability matrix can be checked for validity using `is_valid_markov_matrix()`.

4.3 Information measures

Information measures such as the entropy, conditional entropy, and Kullback-Leibler distance are provided on discrete sources PDF. The entropy $H(X)$ of a discrete source X of cardinal N with PDF $P(X)$ is defined as:

$$H(X) = -\sum_{x=0}^{N-1} P(X = x) \log_2 P(X = x)$$

If $P(X = x)$ is null, it is omitted from the sum (which stems from the limit of $x \log x$). It represents the average number of bits per sample needed to represent a realization of the source. The entropy of a PDF (expressed as a vector), is computed using the `entropy()` function. The special case of the binary source can be computed more simply by providing the probability of the zero bit and calling `entropy_bin()`.

The conditional entropy $H(X|Y)$ of a discrete N-valued random variable X knowing another discrete M-valued random variable Y given the joint PDF matrix $P(X, Y)$ is defined as:

$$H(X|Y) = -\sum_{x=0}^{N-1} \sum_{y=0}^{M-1} P(X = x, Y = y) \log_2 P(X = x|Y = y)$$

In Libit, only the special case of a random markov source is currently supported, where $N = M$ and given the transition matrix $P(X|Y)$, the joint PDF is estimated by first decomposing $P(X, Y)$ into $P(X|Y) \cdot P(Y)$ and obtaining $P(Y)$ from the Froebenius theorem. This conditional entropy can be computed using `entropy_markov()` and providing the matrix of transition probability $P(X|Y)$.

The Kullback-Leibler distance or relative entropy is defined between two PDF $P(X)$ and $P'(X)$ as:

$$d_K(P(X), P'(X)) = \sum_{x=0}^{N-1} P(X = x) \log_2 \frac{P(X = x)}{P'(X = x)}$$

It corresponds to the excess rate needed for coding the source described by P(X) using the PDF P'(X) instead of the appropriate PDF P(X). Strictly speaking this measure is not a distance, as it is not symmetric. It can be computed using the function `vec_distance_kullback_leibler()`.

Program 4.26: Information measure example

```
/* The probability distribution function of the source        */
vec pdf = vec_new_string( "0.6 0.4" );          /* pdf = [ 0.6 0.4 ]        */

/* a conditional PDF for a markov source (same stationary law as above)      */
mat cpdf = mat_new(2, 2);
cpdf[0][0] = 2.0/3.0; /* P(X(t)=0 | X(t-1)=0) */
cpdf[1][0] = 1.0/3.0; /* P(X(t)=1 | X(t-1)=0) */
cpdf[0][1] = 1.0/2.0; /* P(X(t)=0 | X(t-1)=1) */
cpdf[1][1] = 1.0/2.0; /* P(X(t)=1 | X(t-1)=1) */
                                                 /* cpdf = [ [ 0.667 0.5 ]
                                                             [ 0.333 0.5 ] ] */

/* compute the entropy */
double H  = entropy(pdf);                        /* H = 0.971 */
double Hb = entropy_bin(0.6);                    /* same thing using P(X=0)  */

/* compute the conditional entropy */
double Hc = entropy_markov(cpdf);                /* Hc = 0.951 < H */


/* a uniform PDF */
vec uni = vec_new_set(2, 1.0/2.0);               /* uni = [ 0.5 0.5 ]        */

/* compute the relative entropy */
double r = vec_distance_kullback_leibler( pdf, uni );   /* r = 0.029 = 1 - H */
```

Chapter 5

Source Coding Tools

Here is a description of the various source coding tools provided by the library.

5.1 Random numbers and source generation

The library provides various source generation functions. Memoryless sources include discrete sources such as a binary source, and a generic discrete source generator from the discrete PDF (probability density function). Memoryless continous sources include a uniform source, a gaussian source and a generic continuous source generator from the continuous PDF. No correlated source generators are provided yet.

5.1.1 Random number generator

The pseudo-random number generator provided by libit is based on the MT19937 algorithm. It has a large period of 10^{623} which is much better than the standard C function `rand()`. It should *not* be used for cryptography without proper secure hashing of the generated words. For more information on this generator, see M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator", ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3–30.

A random number uniformly distributed in [0,1) is generated using the function `it_rand()`, which returns a double. Note that the resolution of this number is actually 32-bit (therefore the term 'uniform' is not strictly correct). The random number generator need not be seeded if you want to produce the same sequence of pseudo-random numbers at each execution. Otherwise, the function `it_seed()` is provided which initializes the random number generator with a given 32-bit seed. The generator can also be initialized from the system clock using `it_randomize()`. Keep in mind that all these initialization procedures are *not secure* enough for any cryptographic use.

Program 5.27: Random number generation example

```
/* initialize the pseudo-random generator from the system clock */
it_randomize();

/* generate a random number uniformly distributed in [0,1) */
double s = it_rand();
```

Non-uniform random number generation is provided in the form of a normal (Gaussian) generator and a generic generator from the probability distribution function. The normal generator returns a sample of a normally distributed source, that is a Gaussian source of zero mean and variance one. The method used for generating such a sample is the Box-Mueller method.

Program 5.28: Normally distributed random number generation example

```
/* initialize the pseudo-random generator from an integer */
it_seed(0xdeadbeef);

/* generate a normally distributed random number */
double s = it_randn();
```

The generic random number generator `it_randpdf()` returns a sample of a source distributed according to the specified PDF. This PDF is defined using an it_function, assuming it is null outside a specified range and that it's maximum value is taken in zero. For more information on functions, see Section 2.4. The method used to generate a sample is the acceptance-rejection method.

Program 5.29: Generic random number generation example

```
/*-- probability density function declaration --*/

/* parameters of the Laplacian distribution */
it_function_args(laplacian_pdf) {
  double lambda;
};
/* Laplacian probability density function definition (symmetric) */
it_function(laplacian_pdf)
{
  double lambda = it_this->lambda;

  return(1. / (2.*lambda) * exp(-fabs(x) / lambda));
}
```

```
/*-- generate a random number according to this PDF --*/

/* declare the Laplacian parameters */
it_function_args(laplacian_pdf) laplacian_args;

/* set the Laplacian parameter lambda to 1/sqrt(2) (variance equal to 1) */
laplacian_args.lambda = 1/sqrt(2.0);

/* generate laplacian source of 20 samples, assuming the distribution is */
/* null outside the range [-10,10]                                      */
double s = it_randpdf(-10, 10, laplacian_pdf, &laplacian_args);
```

5.1.2 Discrete sources

To generate a binary random vector, the function `source_binary()` is provided. It creates a random bvec of zeroes and ones according to the probability of the zero bit. The generic memoryless discrete source generator takes a vector representing the probability of each symbol. The length of the vector corresponds to the number of discrete symbols, where each element represents the probability of drawing the corresponding symbol. Therefore, this PDF vector must sum to one. The random vector is generated by drawing a uniform random number and taking the inverse of the cumulative density function.

The following example shows how to generate such random vectors.

Program 5.30: Discrete random source generation example

```
/* generate a binary source of 20 samples with more zeroes than ones */
bvec B = source_binary(20, 0.7);

/* generate a 4-ary discrete source of 20 samples according to the PDF */
vec pdf = vec_new_string("0.5 0.3 0.1 0.1"); /* probability density function */
S = source_memoryless(20, pdf );
```

5.1.3 Continuous sources

A vector of uniformly distributed samples taken in the range [a,b) is generated using the function `source_uniform()` and specifying the bounds a and b. Similarly, a vector of normally (Gaussian) distributed samples is generated using `source_gaussian()` and specifying the mean and variance of the distribution. The following example shows how to generate these random vectors.

Program 5.31: Uniform and Gaussian random source generation example

Object	Quantizer
it_quantizer_t	Vector quantizer
it_scalar_quantizer_t	Scalar quantizer
it_uniform_quantizer_t	Uniform scalar quantizer
it_trellis_coded_quantizer_t	Trellis coded quantizer

Table 5.1: Quantizers

```
/* generate a uniform source of 20 samples distributed in [0,1] */
vec v = source_binary(20, 0, 1);

/* generate a gaussian source of 20 samples with mean 0 and variance 1 */
vec g = source_gaussian(20, 0, 1);
```

Other memoryless random sources can be generated by providing the PDF as an it_function and calling `source_pdf()`. For more information on functions, see Section 2.4. The pdf must be centered on its maximum value and bounds must be provided where the PDF may be assumed to be zero. The following example shows how to draw a random memoryless vector from a Laplacian distribution, using the function definition from Example 5.32.

Program 5.32: Generic random source generation example

```
/* declare the Laplacian parameters */
it_function_args(laplacian_pdf) laplacian_args;

/* set the Laplacian parameter lambda to 1/sqrt(2) (variance equal to 1) */
laplacian_args.lambda = 1/sqrt(2.0);

/* generate laplacian source of 20 samples, assuming the distribution is */
/* null outside the range [-10,10]                                       */
source = source_pdf(20, -10, 10, laplacian_pdf, &laplacian_args);
```

5.2 Quantization

Various quantizers are provided in an object oriented hierarchy. All derive from the it_quantizer_t object, which is the vector quantizer. Currently, the quantizers described in Table 5.2 are defined.

A value is quantized to an index with the `it_quantize()` call. A value is dequantized using the `it_dequantize()` call. A whole real-valued vec can be quantized to an ivec using the `it_quantize_vec()` call, and dequantized with the `it_dequantize_vec()` call. Similarly matrices can be quantized and dequantized row by row with the `it_quantize_mat()` and `it_dequantize_mat()` calls.

The default scalar quantizer can be initilized from a codebook. Quantization is done using a logarithmic search inside that codebook for the closest codeword. Dequantization done in a much faster way by simply indexing the codebook. The `lloyd_max()` call can be used to compute the optimal codebook of source (minimizing the quantization distortion for a given fixed rate). Here is an example on how to build the Lloyd-Max quantizer with 4 levels for the gaussian source of zero mean and variance 1. We use the gaussian function defined in Section 2.4. For more information on functions, refer to that section.

Program 5.33: LLoyd-Max quantization example

```
/* set the standard deviation of the gaussian to 1 */
gaussian_args.sigma = 1.0;
/* generate the Lloyd-Max quantization codebook for 4 levels */
/* The pdf is limited to the range [-5,5] (assumed equal to 0 outside) */
codebook = lloyd_max(gaussian, &gaussian_args, -5, 5, 4);

/* create the associated quantizer */
it_scalar_quantizer_t quantizer = it_scalar_quantizer_new(codebook);

/* generate a random vector distributed normally */
source = source_gaussian(100000, 0, gaussian_args.sigma);

/* quantize */
ivec qsource = it_quantize_vec(quantizer, source);

/* dequantize */
vec rec = it_dequantize_vec(quantizer, qsource);
```

The uniform scalar quantizer can be initialized either from the position of the center and the step, or from the expected range of the values and the number of quantization levels. The first version uses the `it_uniform_quantizer_new_from_center()` call and can be given a factor to have a larger central zone (dead-zone).

The second versions is created using the `it_uniform_quantizer_new_from_range()` call. The uniform scalar quantizer uses a division for quantization which is much faster than the generic logarithm search algorithm. Here is an example on how to use each version:

Program 5.34: Uniform scalar quantization example

```
/* create a random vector uniformly distributed in [0,1] */
vec source = source_uniform_01(100000);

/* create a new uniform quantizer with 256 levels
   equally spread in the [0,1] range              */
it_uniform_quantizer_t uniq =
    it_uniform_quantizer_new_from_range(256, 0.0, 1.0);

/* quantize / dequantize */
ivec qsource = it_quantize_vec(uniq, source);
vec rec = it_dequantize_vec(uniq, qsource);

/* create a new uniform quantizer with step 1/256 centered on 0
   with a dead-zone of twice the step size                      */
it_uniform_quantizer_t uniq =
    it_uniform_quantizer_new_from_center(0, 1/256., 2.0);

/* quantize / dequantize */
ivec qsource = it_quantize_vec(uniq, source);
vec rec = it_dequantize_vec(uniq, qsource);
```

The trellis-coded quantizer (TCQ) is built from any scalar quantizer (uniform or generic) and a convolutional code. TCQ is a fast vector quantization technique achieving near optimal performance. It is based on the set partitioning by a convolutional code of the product codebook of a scalar quantizer. For more information on TCQ, refer to ”M.W. Marcellin and T.R. Fisher, Trellis-coded quantization of memoryless and gauss-markov sources, IEEE Trans. Comm., 38:82–93, Jan. 1990”. Here is an example on how to define and use such a quantizer:

Program 5.35: TCQ example

```
/* generator polynomials for the convolutional code (rate 1/2) */
imat generator = imat_new(1, 2);
generator[0][0] = 0133;
generator[0][1] = 0171;
/* create a convolutional code */
it_convolutional_code_t *code = it_convolutional_code_new(generator);

/* create a uniform quantizer with 512 levels */
uniq = it_uniform_quantizer_new_from_range(512, 0.0, 1.0);
/* create a TCQ quantizer from the uniform quantizer and the convolutional code */
tcq = it_trellis_coded_quantizer_new_partition(code, uniq);
```

Method	Function name	Description
Fixed length Code	vlc_flc	All the codewords of the same size (lexicographic)
Huffman	vlc_huffman	The optimal Variable Length Code
Hu-Tucker	vlc_hu_tucker	The optimal lexicographic Variable Length Code
Generic	vlc_read	Custom code entered by the user as a string

Table 5.2: Variable Length Codes construction methods

```
/* create a random vector uniformly distributed in [0,1] */
vec source = source_uniform_01(100000);

/* quantize / dequantize */
ivec qsource = it_quantize_vec(tcq, source);
vec rec = it_dequantize_vec(tcq, qsource);
```

5.3 Variable Length Codes

In the following, the term Variable Length Code (VLC) stands for a scalar variable length code over a finite alphabet. Although strictly speaking, arithmetic codes are variable length codes, arithmetic codes are not covered by this term. A variable length code is defined as a set of binary codewords. Such a code is a prefix code, i.e. a given codeword can not be the prefix of another.

In the library, the typename for Variable Length Codes objects is vlc_t. The users should only manipulate pointers vlc_t *. The set of supported construction methods for Variable Length Codes is given below. All these methods return an allocated vlc_t pointer that allows to handle the corresponding object.

The variable length encoder takes an input vector of symbols and outputs a vector of bits. The symbols are represented by integers between 0 and $N-1$, where N is the number of symbols handled by the variable length code. The function `vlc_encode_concat()` is used for the encoding. It, takes the input vector of integers (type ivec) and produces a bitstream represented by a byte vector (bvec). The decoder takes such a bitstream as an input and allows to recover the original vector. It is handled with function vlc_decode_concat.

Here is an example on how to define and use a variable length code:

Program 5.36: VLC example

```
/* The probability distribution function of the source        */
vec pdf = vec_new_string( "0.5 0.3 0.1 0.1" );
```

```
/* Create an Huffman code optimum for the source pdf          */
vlc_t * vlc = vlc_huffman( pdf );

/* Generate a sequence according to the given probability law */
int N = 20;
ivec S = source_memoryless( N, pdf );

/* Encode and decode the sequence S with the Huffman Code     */
bvec B = vlc_encode_concat( vlc, S );
ivec D = vlc_decode_concat( vlc, B );

/* Print the vectors S, B and D                               */
it_printf( "S = $d\nB = $b\nD = $d\n", S, B, D );

/* Print the mean description length of the code and the
   observed average description length                        */
printf( "Mdl            = %lf\n", vlc_mdl( vlc, pdf ) );
printf( "Obs. av. length = %lf\n",
          bvec_length( B ) / (double) ivec_length( S ) );

/* Delete the code and defines an user defined code such that
   its Kraft sum not equal to 1 . Print the code.             */
vlc_delete( vlc );
vlc = vlc_read( "{0 11 101 1001}" );
printf( "Kraft sum       = %lf\n", vlc_kraft_sum( vlc ) );
printf( "Custom Code     = " );
vlc_print( vlc );
printf( "\n" );

vlc_delete( vlc );
ivec_delete( S );
ivec_delete( D );
bvec_delete( B );

/*-------------------------------------------------------------
   The program generate something like that:

S = [0 2 0 1 0 1 2 3 1 0 0 0 0 0 2 0 0 1 3 2]
B = [0 1 0 0 0 1 1 0 1 1 1 0 0 1 0 1 1 1 0 0 0 0 0 1 0 0 0 0 1 1 1 0 1 1 0 0]
D = [0 2 0 1 0 1 2 3 1 0 0 0 0 0 2 0 0 1 3 2]
Mdl             = 1.700000
Obs. av. length = 1.800000
Kraft sum       = 0.937500
```

```
  Custom Code      = {0 11 101 1001}                              */
```

Note that the users should not handle directly vlc_t objects but rather the corresponding pointers. The memory allocated by these functions should be freed with the function `vlc_delete()`.

Chapter 6

Channel Coding Tools

This chapter presents the various channel coding tools provided by libit.

6.1 Modulation

Modulation consists in encoding a binary vector representing the message into a vector representing the physically emitted message. Currently only binary phase shift keying (BSPK) modulation is provided.

BPSK modulation consists in coding each bit of the message with either a +1 (for bit 1) or a -1 (for bit 0) value.

Program 6.37: BPSK modulation example

```
/* generate a random message of 10 equiprobable bits */
bvec message = source_binary(10, 0.5);
/* modulate the message using BPSK */
vec modulated = modulate_bpsk(output);
```

6.2 Channels

Channels are used to simulate transmission noise. Currently the additive white gaussian noise (AWGN) and binary symetric (BSC) channels are defined.

The binary symetric channel is used simply by knowing the cross-over probablity, that is the probability for a bit to be inverted during transmission. To simulate transmission over a BSC channel, use the `channel_bsc()` function.

Program 6.38: BSC channel example

```
/* generate a random message of ten equiprobable bits */
bvec message = source_binary(10, 0.5);

/* send over a BSC channel of cross-over probability 0.1 */
bvec received = channel_bsc(message, 0.1);
```

The AWGN channel is parametrized by the standard deviation of the additive white noise added during transmission. To simulate transmission over an AWGN channel, use the `channel_awgn()` function.

Program 6.39: AWGN channel example

```
/* generate a random message of ten equiprobable bits */
bvec message = source_binary(10, 0.5);
/* modulate the message using BPSK (0 => -1, 1 => +1) */
vec modulated = modulate_bpsk(output);
/* send over an AWGN channel of standard deviation 0.5 */
vec received = channel_awgn(message, 0.5);
```

6.3 Convolutional Codes

Convolutional codes of rate $k : n$ are provided using the it_convolutional_code_t object type. They are initialized from the matrix of generator polynomials, with each row representing an input bit and each column an output bit. Sequences are encoded using the `it_convolutional_code_encode()` call. Viterbi decoding is provided through the `it_viterbi_decode()` provided the metrics are given. These metrics are represented using matrix of 2^n columns corresponding to branch labels (the n bits output by the convolutional code) and a number of rows equal to the length of the sequence. The Viterbi algorithm returns the sequence of maximal total metric. Here is an example declaring and using a 133/171 convolutional code of rate one half:

Program 6.40: Convolutional code example

```
/* the matrix of generator polynomials */
imat generator = imat_new(1, 2);
generator[0][0] = 0133; /* generator polynomial using the C octal notation */
generator[0][1] = 0171;

/* create the convolutional code */
it_convolutional_code_t *code = it_convolutional_code_new(generator);
```

```
/* generate a random binary sequence of 100 bits */
bvec input = source_binary(100, 0.5);

/* encode the sequence */
output = it_cc_encode(code, input);

/* modulate */
vec modulated = modulate_bpsk(output);

/* transmit over an AWGN channel of variance 1/4 */
vec received = channel_awgn(modulated, 0.5);

/* compute metrics */
mat metrics = bspk_metrics(cc, received);

/* decode */
bvec decoded = it_viterbi_decode(cc, metrics);
```

Chapter 7

Signal Processing Tools

This chapter presents the various signal processing tools provided by the library.

7.1 1D Transforms

All transforms in LIBIT share the same framework. They are objects taking a generic vector (Vec) as input and output a transformed generic vector, which may be of different type or length (e.g. for redundant transforms). Declaring new transforms is done by inheriting from the it_transform_t object and defining the transform and inverse transform methods. For more details on objects and object oriented programming in C with libit, see Chapter A. Here is the set of transforms provided by libit (which is currently quite limited).

7.2 Discrete Fourier Transform

The Fourier transform of a signal gives a frequency representation of the energy and relative phase present in this signal. The discrete transform X of a signal x is defined in complex domain as:

$$X[n] = \sum_{k=0}^{N-1} x[k] e^{-2i\pi \frac{kn}{N}}.$$

This transform is invertible and the original signal can be recovered by using the inverse transform:

$$x[k] = \frac{1}{N} \sum_{n=0}^{N-1} X[n] e^{2i\pi \frac{kn}{N}}.$$

In libit, the discrete Fourier transform of a vector is obtained by calling `it_dft()`, which takes a complex vector (cvec) as input and returns a complex vector representing the amplitude and phase for each frequency. The inverse transform is obtained by using `it_idft()`.

Currently the transform is implemented as in the math formula, which is inefficient ($O(N^2)$ vs $O(N \log N)$ for the FFT algorithm). However it has the advantage being able to handle vectors of any length. For vectors where the length can be decomposed in a power of two times an odd number, the current implementation of the DFT could be optimized much. Also, there is no optimized function for the special case of real samples yet; Thus, real-valued vectors (vec) must be converted to complex vectors using `vec_to_cvec()` before calling `it_dft()` (which will lead to a symmetric conjugate frequency representation).

Program 7.41: Discrete Fourier transform example

```
/* analyse the vector using the discrete Fourier transform */
cvec vt = it_dft(v);

/* recompose the vector by inverse transform */
cvec vr = it_idft(vt);
```

7.3 Discrete Wavelet Transform

The dicrete wavelet transform (DWT) analyses a signal into a given number of subbands (scales) while maintaining a time-frequency localization compromise. High frequencies and impulses are well localized in space, while low frequencies are precise. The DWT provided by libit is a dyadic, reversible transform implemented using the lifting scheme. Currently there is no factorization algorithm in the library to extract the lifting steps from the definition of the FIR filters. Therefore these steps must be given explicitly. However, the most commonly used 5/3 and 9/7 biorthogonal wavelet lifting steps are provided.

To apply a wavelet transform on a vector, use the `it_dwt()` function. It takes a real vector as input and output a real vector of the same length containing all the concatenated subbands (from low-frequency to high-frequency). Additional parameters include the number of levels and the lifting steps used for the decomposition. The output vector can be split in an array of subbands using `it_wavelet_split()`, and merged back using `it_wavelet_merge()`. The inverse transform is provided through `it_idwt()`. Here is an example on how to analyse a signal using the 9/7 biorthogonal wavelet.

Program 7.42: Discrete wavelet transform example

```
/* analyse the vector using a 5-level 9/7 wavelet decomposition */
vec vt = it_dwt(v, it_wavelet_lifting_97, 5);
```

```
/* clear the high subband */
vec_set_between(vt, (vec_length(v) + 1) / 2, end, 0);

/* recompose the vector by inverse transform */
vec vr = it_idwt(vt, it_wavelet_lifting_97, 5);
```

7.4 2D Transforms

Similar to the 1D case, 2D transforms share the same framework. They are objects taking a generic matrix (Mat) as input and output a transformed generic matrix, which may be of different type, width or height as the original matrix (e.g. for redundant transforms). Declaring new 2D transforms is done by inheriting from the it_transform2D_t object and defining the transform and inverse transform methods. For more details on objects and object oriented programming in C with libit, see Chapter A. Here is the set of 2D transforms provided by libit (which is currently quite limited too).

7.5 Generic Separable 2D Transform

A generic 2D separable transform is provided. This transform takes a 1D transform and applies it on the row, then the columns of the input matrix. Here is an example on how to perform a 2D Fourier transform by combining the 1D Fourier transform with the separable transform.

Program 7.43: Separable transform example

```
/* create a Fourier transform */
it_fourier_t fourier = it_fourier_new();
/* create a separable transform out of it */
it_separable2D_t fourier2D = it_separable2D_new(fourier);

/* apply the 2D discrete Fourier transform on a matrix */
cmat mt = (cmat) it_transform2D(fourier2D, (Mat) m);

/* inverse the 2D discrete Fourier transform */
cmat mr = (cmat) it_itransform2D(fourier2D, (Mat) mt);
```

7.6 Separable 2D Wavelets

pro:separable2dwavelets

Bidimensional separable wavelet decomposition is provided using the lifting implementation. The lifting procedure guarantees perfect reconstruction and provides an efficient way of filtering. Currently only the 9/7 and 5/3 wavelets are provided.

A new 2D wavelet transform is created from the lifting steps, the width and height of the image to transform and the number of decomposition levels. As a particular kind of 2D transform, transform and inverse transform are provided using the it_transform2D() and it_itransform2D() calls. Here is an example using the 9/7 wavelet:

Program 7.44: 2D wavelet transform example

```
/* read your favorite version of lena */
mat m = mat_pgm_read("lena.pgm");

/* create a new 9/7 wavelet object for a decomposition on 5 levels */
it_wavelet2D_t *wavelet2D = it_wavelet2D_new(it_wavelet_lifting_97, 5);

/* transform the image */
mat mt = it_transform2D(wavelet2D, m);

/* reconstruct the image */
mat mr = it_itransform2D(wavelet2D, mt);
```

Appendix A

Objects

For high level components, an object oriented approach was taken to allow for a greater flexibility through the inheritance and polymorphism concepts. This appendix gives some details on the usage and implementation of objects for advanced users.

A.1 Declaration, construction and destruction

All object types are of the form it_objectname_t. All objects are deleted using the expression `it_delete(x)` where x is an object.

Program A.45: Object

```
/* declare a scalar quantizer object */
it_scalar_quantizer_t *scalar_quantizer;

/* create and initialize a new scalar quantizer centered on 0 with step 1 */
scalar_quantizer = it_scalar_quantizer_new_from_center(0, 1);

/* delete the scalar_quantizer object */
it_delete(scalar_quantizer, v);
```

A.2 Casting

Casting is done using macros of the form 'IT_OBJECTNAME(x)', which cast the object 'x' to the type 'it_objectname_t'. If the object cannot be cast properly to one of its parent types, an error will be displayed at runtime.

Program A.46: Casting

```
/* declare a quantizer object */
it_quantizer_t *quantizer;

/* cast the object 'scalar_quantizer' to its parent type 'it_quantizer_t' */
quantizer = IT_QUANTIZER(scalar_quantizer);
```

A.3 Polymorphism

Polymorphism allows to use an object in a place where a parent object is expected. The child object is seen as the parent object except the methods of the child class are called, rather than the methods of the parent class.

Program A.47: Polymorphism

```
it_scalar_quantizer_t *scalar_quantizer; /* declare a scalar quantizer object */
it_quantizer_t *quantizer; /* declare a quantizer object */
vec v = vec_new_ones(10);  /* a vector */

/* create and initialize a new scalar quantizer centered on 0 with step 1 */
scalar_quantizer = it_scalar_quantizer_new_from_center(0, 1);

/* cast the it_scalar_quantizer_t object to a it_quantizer_t object. */
/* valid since it_quantizer_t is the parent of it_scalar_quantizer_t */
quantizer = IT_QUANTIZER(scalar_quantizer);

/* call a method of the it_quantizer_t class on the quantizer. */
/* the method called is that of it_scalar_quantizer_t due to polymorphism. */
it_quantize(quantizer, v);

/* this does the same thing as above, with a simpler syntax. */
it_quantize(scalar_quantizer, v);
```

A.4 Comments on implementation

The object orientated mechanisms in libit are greatly inspired from Glib/GTK `http://www.gtk.org/`. Objects are implemented as structures. Methods are implemented as functions pointers inside the object structure. Inheritance is achieved by placing the expression 'it_extends(it_parent_t)' where it_parent_t is the type of the parent object. All objects inherit from the type 'it_object_t'.

Dynamic type checking is achieved by assigning a unique identifier (UID) to each object type and keeping track of the inheritance structure.