The BeanPolE library

(BoolEAN POLynomial Evaluation)

by Charles Bouillaguet

charles.bouillaguet@lip6.fr

This manual describes how to use the BeanPolE library.

Both the code of the library and this manual are in the public domain.

1 Introduction

BeanPolE is a portable library written in C for manipulating multivariate Boolean polynomials in dense representation. It notably implements two efficient algorithms to evaluate a Boolean polynomial on all possible inputs.

BeanPolE stands for BoolEAN POLynomial Evaluation.

The library comes with demonstration programs and a test suite. We try to make sure that it works correctly.

The latest version of the library is available at

`https://gitlab.lip6.fr/almasty/beanpole`

Please contact Charles Bouillaguet (`charles.bouillaguet@lip6.fr`) to report bugs, suggestions for improvements or report interesting uses of the library.

1.1 How to use this Manual

Everyone should read Chapter 2 [BeanPolE Basics], page 2, before using the library.

2 BeanPolE Basics

This chapter covers the information that is common to all functions of the library.

2.1 Headers and Libraries

All declarations needed to use the BeanPolE library are collected in the include file `beanpole.h`.

All programs using BeanPolE must add `beanpole.c` to their list of sources.

The library is quite concise (about 700 lines of verbose code), therefore we did not deem it necessary to make it a "proper" library using libtool and the likes.

2.2 Initialization

Before being used, the library must be initialized. Calling any other function before doing so is an error and leads to undefined behavior.

`void beanpole_init()` [Function]

Initialize the library.

This function is idempotent (i.e., it is harmless to call it several times). Its first invocation populates the global arrays of the library and terminates in $O(BEANPOLE_MAXN^2)$ operations, which is completely negligible. Subsequent invocations of this function return in constant time.

2.3 Naming and Argument Order Conventions

Variables are numbered from 0 to $n - 1$.

All functions start with the prefix `beanpole_`.

Arguments appear in the following order:

- data access function and opaque argument for this function;
- number of variables (n);
- degree (d);
- monomial order ($order$);
- array of coefficients (A);
- iterator variable (it).

Read-only arguments come first and modified arguments come last.

Functions do an extremely limited level of error checking on their arguments. They simply crash the program if incorrect values are given.

2.4 Memory Usage and Hardcoded Limits

The library does not dynamically allocate any memory on the heap (it doesn't call `malloc()` or any other equivalent function), and it only uses the stack parsimoniously. This makes it even more obvious that the algorithms work in-place.

This requires a few (read-only) global arrays of modest size. Therefore, there are compile-time bounds on the number of variables and the degree of polynomials that can be handled. These bounds are easy to change by altering the corresponding preprocessor constants at the begining of `beanpole.h`.

`BEANPOLE_MAXN` [Variable]
Maximum number of variables that the library handles (default to 128).

`BEANPOLE_MAXD` [Variable]
Maximum degree that the library handles. (default to 16).

These are defined in `beanpole.h` as follows:

```
#define BEANPOLE_MAXN  128                    /* maximum #variables */
#define BEANPOLE_MAXD   16                       /* maximum degree */
```

Invoking any library function with `n > BEANPOLE_MAXN` or `d > BEANPOLE_MAXD` yields a fatal error.

2.5 Monomial Orders

The library handles two monomials orders: the colexicographic order and the graded colexicographic order. They are slightly uncommon, but lead to much simpler programming. We formally define them below.

In the library, monomial orders are described by variables of type `beanpole_order`, which is in fact an `enum` containing the two constants `BEANPOLE_COLEX` and `BEANPOLE_GRCOLEX`.

2.5.1 Formal Definitions

Suppose that we work with n variables denoted by $x_0, x_1, x_2, ...$ A monomial can be written $x_0^{e_0} x_1^{e_1} ...$ and is completely described by the "exponent vector" $(e_0, e_1, e_2, ...)$.

The usual *lexicographic order* over monomials is obtained by ordering these exponent vectors in... lexicographic order. This yields:

$$1 < x_7 < x_4x_5 < x_1 < x_0 < x_0x_7 < x_0x_1$$

All monomials that contain x_0 (the first variable) are greater than monomials that do not.

The *colexicographic order* (abbreviated colex) is obtained by ordering the exponent vectors in lexicographic order while reading them from right to left (as opposed to reading them from left to right as usual). This is the same thing as the usual lexicographic order with variables in reversed order $(..., x_2, x_1, x_0)$. This yields:

$$1 < x_0 < x_1 < x_0x_1 < x_4x_5 < x_7 < x_0x_7$$

All monomials that contain x_7 (the last variable) are greater than monomials that do not. More formally, $s < t$ iff there exists $0 <= i < n$ such that the last $i-1$ exponents of s and t are equal but the i-th exponent of s is less than the i-th exponent of t.

The colex order is called `invlex` in SageMath and `rp` in Singular.

In the *graded colexicographic order*, monomials are ordered by increasing degree, and ties are broken using the colexicographic order. This yields

$$1 < x_0 < x_1 < x_7 < x_0x_1 < x_4x_5 < x_0x_7$$

This monomial order is not available in other computer algebra systems, to the best of our knowledge. It is *not the same* as the more common "Degree Reverse Lexicographic" order favored by Groebner basis algorithms.

In any case, the constant monomial has rank zero in both the colex and the graded colex order.

2.6 Data Structures

2.6.1 Monomial Representation

A Boolean monomial is a subset of $\{0, 1, ..., n-1\}$. Functions receive and produce monomials in *sparse* representation: a degree-d monomial is an array of d `int` (sorted in ascending order) that specifies the variables it contains.

To describe a monomial, one must provide:

- its degree ;
- a pointer to an array containing its variables.

For instance, this code chunk obtains the rank of $x_2x_6x_{11}$ among all degree-5 monomials in 12 variables, in lexicographic order:

```
int m[3] = {2, 6, 11};
size_t r = beanpole_rank(12, 5, BEANPOLE_COLEX, 3, m);
```

2.6.2 Polynomial Representation

BeanPolE uses a simple dense representation. A degree-d polynomial in n variables is stored in an array (usually denoted by A) containing its coefficients. This is a *dense* representation because this array contains the coefficient of *all* monomials in n variables of degree less than or equal to d.

Monomials are naturally numbered according to their rank in a given monomial order (see Section 2.5 [Monomial orders], page 3). To completely specify a Boolean polynomial, it is necessary to provide the following information:

- The number of variables (usually denoted by n).
- The degree (usually denoted by d).
- The monomial order (usually denoted by *order*).
- The array of coefficients (usually denoted by A).

In the array of coefficients, `A[i]` is the coefficient of the `i`-th monomial in n variables, of degree at most d, in the specified monomial order.

It is the user's responsibility to allocate the arrays in which the coefficients are stored. To facilitate this, the library offers a convenience function that returns the number of coefficients of a polynomial.

`size_t beanpole_size(int n, int d)` [Function]

Return the total number of monomials in n variables of degree less than or equal to d.

This function returns in constant time. It returns the correct result even if $n < d$.

Here is an example of typical use:

```
size_t N = beanpole_size(n, d);
bool *A = malloc(N * sizeof(*A));
```

The ranking functions can be useful to populate these arrays (see Section 3.1 [Ranking / Unranking], page 7). The companion example programs illustrate how to parse text representations of polynomials and populate this dense representation.

2.7 Genericity

Functions of the library perform the same sequence of operations regardless of the coefficients of input polynomials (they are "data-oblivious"). Moreover, they only access the coefficients of polynomials through two simple "update" operations: `B[j] ^= A[i]` or `C[k] ^= A[i] & B[j]`, where A, B and C are arrays of coefficients. This makes the library quite generic: it only requires the two update function described above to be provided, and is oblivious to the actual machine representation of the coefficients.

This enables the use of the same library code in a variety of contexts: polynomials represented by simple array of `bool` values, polynomials represented by a bitvector, several polynomials considered as a single polynomial with vector coefficients, etc.

All functions that actually access the array of coefficients of a polynomial takes as argument an *updater* function with one of the following signatures:

`void beanpole_add_fn(void *ctx, const void *A, size_t u, void *B, size_t v)` [Function]

`void beanpole_addmul_fn(void *ctx, const void *A, size_t u, const void *B, size_t v, void *C, size_t w)` [Function]

Do `B[v] ^= A[u]` and `C[w] ^= A[u] & B[v]`, respectively.

ctx is an opaque argument provided by the user to the functions of the library, and which is then piggybacked to these "update" functions.

For instance, here is a possible way to implement the "updater" function for arrays of `bool` values:

```
void bool_add(void *, const void *_A, size_t i, void *_B, size_t j)
{
        const bool *A = _A;
        bool *B = _B;
        B[j] ^= A[i];
}
```

The *ctx* argument enables the user to provide run-time information (e.g., not known at compile-time) to the "updater" function, without resorting to global variables. One possible such use consists in dealing with vectors of coefficient of unknown length. Here is an example:

```
vector_add(void *ctx, const void *_A, size_t i, void *_B, size_t j)
{
        int w = *((int *) ctx);                 /* vector length */
        const uint64_t *A = _A;
        uint64_t *B = _B;
        for (int k = 0; k < w; k++)         /* sum the w entries */
                B[w*j + k] ^= A[w*i + k];
}
```

As an illustration , we give below the code of a simple function (not part of the library) that sums two polynomials. This amounts to sum their coefficients. Access to their coefficient is done through the user-provided "updater" function.

```
/* B += A */
void polynomial_sum(beanpole_add_fn updater, void *ctx,
                    int n, int d, const void *A, void *B)
{
```

```
        size_t N = beanpole_size(n, d);
        for (size_t i = 0; i < N; i++)
                updater(ctx, A, i, B, i);     /* B[i] ^= A[i] */
}
```

2.8 Iterators

In order to iterate over a sequence of objects (entries of the truth table, monomials, etc.), the library provides *iterators* made of three functions:

- `xxx_prepare(...)`: initializes the iteration.
- `xxx_advance(...)`: move to the next element.
- `xxx_finished(...)`: indicates whether there is a next element.

Each of these function take an argument of type `struct beanpole_iterator`. They can be used as follows:

```
struct beanpole_iterator it;
xxx_prepare(..., &it);
while (!xxx_finished(..., &it)) {
        ...
        xxx_advance(..., &it);
}
```

2.9 Thread-Safety

All functions of the BeanPole library are sequential. With the exception of `beanpole_init` (see Section 2.2 [Initialization], page 2), they are all reentrant (they do not mutate a shared global state). Therefore, BeanPolE can safely be used in multithread programs.

However, some function modify their arguments, including the polynomial coefficients. It is therefore the user's responsibility to ensure that conflicting accesses to these values do not occur through concurrent calls to functions of the BeanPole library.

3 Operations over Monomials

This chapter describes functions that operate on monomials in sparse representation (see Section 2.6 [Data Structures], page 4). The rank of a monomial is always a variable of type `size_t`. Monomial orders are specified by a variable of type `beanpole_order` (see Section 2.5 [Monomial orders], page 3).

3.1 Ranking / Unranking

Ranking is the process of determining the rank of a monomial in a given order. Unranking is the process of producing the monomial of a given rank.

`size_t beanpole_rank(int n, int d, beanpole_order order, int k, const int m[])` [Function]

Return the rank, according to the monomial order given in *order*, of the degree-k monomial given by M among all degree-d monomials in n variables.

This function runs in time linear in k.

The ranking function can for instance be used to easily populate the array containing the coefficients of a polynomial.

`int beanpole_unrank(int n, int d, beanpole_order order, size_t rank, int m[])` [Function]

Write in m the monomial of rank *rank* among all degree-d monomials in n variables according to the monomial order specified by *order*. Return its degree.

m must point to a sufficiently large array; up to d integers will be written in m.

This function runs in time linear in n.

3.2 Iteration

The library offers a convenient interface to iterate over all monomials in either the colex or the graded colex order. This makes uses of *iterator* variables of type `struct beanpole_iterator` (see Section 2.8 [Iterators], page 6). They have three fields that describe the current monomial:

- `k`: degree of the current monomial
- `m`: variables of the current monomial
- `rank`: rank of the current monomial

It is not an error to iterate over degree-d monomials in n variables with $n < d$. However it is strictly equivalent to iterating over degree-n monomials in n variables.

`void beanpole_monomials_prepare(int n, int d, beanpole_order order, struct beanpole_iterator *it)` [Function]

Set up the iterator *it* for the iteration over all monomials in n variables of degree at most d in the order given by *order*.

This function runs in constant time.

`bool beanpole_monomials_finished(const struct beanpole_iterator *it)` [Function]

Return `true` when the iteration is finished, i.e. there is no next monomial.

This function runs in constant time.

`void beanpole_monomials_advance(struct beanpole_iterator *it)` [Function]
Advance the iterator to the next monomial.

It is an error to call this function if the iteration is finished. This function runs in amortized constant time.

As an illustration, the two following loops are functionally equivalent, but the second one is much faster.

```
/* bad */
size_t N = beanpole_size(n, d);
for (size_t i = 0; i < N; i++) {
       int m[d];
       int k = beanpole_unrank(n, d, order, m);
       do_stuff(i, k, m);
}

/* good */
struct beanpole_iterator it;
beanpole_monomials_prepare(n, d, order, &it);
while (!beanpole_monomials_finished(&it)) {
        do_stuff(it.rank, it.k, it.m);
        beanpole_monomials_advance(&it);
}
```

3.3 Iteration over fixed-degree monomials

The library also contains an iterator over all degree-d monomials in colex order.

Trying to iterate over degree-d monomials in n variables with $n < d$ does not yield an error; the sequence is nethertheless empty.

WARNING: these functions are mostly intended for internal use; in particular they do not populate the `rank` field of the iterator.

`void beanpole_monomials_degd_prepare(int n, int d, struct beanpole_iterator *it)` [Function]
Set up the iterator *it* for the iteration over all degree-d monomials in n variables in colex order.

This functions runs in time linear in d.

`bool beanpole_monomials_degd_finished(const struct beanpole_iterator *it)` [Function]
Return `true` when there is no next monomial.

This function runs in constant time.

`void beanpole_monomials_degd_advance(struct beanpole_iterator *it)` [Function]
Advance the iterator to the next degree-d monomial.

It is an error to call this function if the iteration is finished. This function runs in amortized constant time.

3.4 Iteration over arbitrary subsets of the variables

While the previous functions enumerate monomials in the n variables $x_0, ..., x_{n-1}$, the following functions allow the iteration over arbitrary subsets of the variables.

`void beanpole_monomials_prepare_general(int n, int d, const int vars[], beanpole_order order, struct beanpole_iterator *it)` [Function]

Set up the iterator *it* for the iteration over all monomials in the n variables given in *vars* of degree at most d in the order given by *order*.

This function runs in constant time.

`void beanpole_monomials_degd_prepare_general(int n, int d, const int vars[], struct beanpole_iterator *it)` [Function]

Set up the iterator *it* for the iteration over all degree-d monomials in the n variables given in *vars* in colex order.

This functions runs in time linear in d.

Here is an example of a typical use-case, enumerating monomials in a subset S of the variables:

```
/* enumerate subsets of {0, ..., n-1} */

/* bad */
beanpole_monomials_prepare(n, d, order, &it);
while (!beanpole_monomials_finished(&it)) {
        /* expand into a subset of S: O(d) penalty per iteration */
        int mm[n];
        for (int i = 0; i < it.k; i++)
                mm[i] = S[it.m[i]];
        do_stuff(it.k, mm);
        beanpole_monomials_advance(&it);
}

/* good (faster) */
beanpole_monomials_prepare_general(n, d, S, order, &it);
while (!beanpole_monomials_finished(&it)) {
        do_stuff(it.k, it.m);
        beanpole_monomials_advance(&it);
}
```

3.5 Multiplication

`int beanpole_monomials_product(int k, const int a[], int l, const int b[], int c[])` [Function]

Compute the product of two monomials of degree k and l respectively, with variables given by a and b. Write the product in c and return its degree.

c must point to a sufficiently large array (up to $k + l$ integers may be written in c). This function runs in time $O(k + l)$.

4 Evaluation

4.1 Single-point Evaluation

The following function allows the evaluation of a Boolean polynomial on an arbitrary input. It takes as argument an "updater" function pointer to access the coefficients of the polynomial (see Section 2.7 [Genericity], page 5).

`void beanpole_eval(beanpole_add_fn` *`updater`*`, void *`*`ctx`*`,` [Function]
int *n*, int *d*, beanpole_order *order*, const void **A*, const bool *x*[], void **y*)

Do `y ^= f(x)`, where *f* is the degree-*d* polynomial in *n* variables whose coefficient are stored in *A* according to the monomial order *order*.

x must be an array of size at least *n*.

In the worst case, the running time of this function is proportional to *d* times the size of the polynomial. In the average case, it runs in time proportional to $d^2 2^{-d}$ times the size of the polynomial.

The implementation of this function is likely suboptimal. However, it is meant as a convenience, and it is not the real focus of the library.

5 Substitution

5.1 Last-variable Flip

`void beanpole_fliplastvar_colex(beanpole_add_fn` *`updater`*`, void *`*`ctx`*`, int` *`n`*`, int` *`d`*`, void *`*`A`*`)` [Function]

Given a polynomial f in n variables of degree d, with coefficients given in the array A in *colex* order, overwrite A with the coefficients of $f(x_0, x_1, ..., x_{n-2}, x_{n-1} + 1)$.

This operation is involutive.

This function runs in time proportionnal to the number of monomials of degree at most $(d-1)$ in $(n-1)$ variables.

This function can be used to fix the last variable of a polynomial with coefficients in *colex* order.

Setting the last variable to zero is easy: because all monomials that contain the last variable appear last, it is sufficient to forget about them and to consider that there is one less variable.

Setting the last variable to 1 can be achieved by "flipping" the last variable using the above function before fixing the last variable to zero (i.e., ignoring it).

Using this technique several times enables to fix the last variables efficiently. This function is used internally in the in-place Moebius transform (see Chapter 8 [Moebius], page 14) and is exposed as a convenience.

5.2 General Variable Substitution

`void beanpole_subs(beanpole_addmul_fn` *`updater`*`, void *`*`ctx`*`,` [Function]
int *n*, int *i*, int *dA*, beanpole_order *orderA*, const void **A*, int *dB*, beanpole_order *orderB*, const void **B*, int *dC*, beanpole_order *orderC*, void **C*);

A, *B* and *C* are polynomials in *n* variables of variables of degree *dA*, *dB* and *dC*, respectively with coefficients in the orders given by *orderA*, *orderB* and *orderC*.

Replace the *i*-th variable by *B* inside *A* and adds the result to *C*. In other terms, do:

$$C := C + A(x_0, \ldots, x_{i-1}, B(x), x_{i+1}, \ldots, x_{n-1}).$$

The input polynomials *A* and *B* are not modified. Calling this function with either `C == A` or `C == B` leads to undefined behavior.

This function runs in time less than to dC times the size of *C*.

Using a polynomial *B* of degree zero is equivalent to fixing the *I*-th variable to a constant value.

This is one of the few functions that takes a user-provided "add-mul" access function.

6 Polynomial Multiplication

The library offers a naive (quadratic) implementation of polynomial multiplication. This has been implemented because it was easy given the availability of the rest of the infrastructure. These are part of the few functions that takes a user-provided "add-mul" access function.

`void beanpole_product_pxp(beanpole_addmul_fn` *`updater`*`, void *`*`ctx`*`, int` *`n`*`, int` *`dA`*`, beanpole_order` *`orderA`*`, const void *`*`A`*`, int` *`dB`*`, beanpole_order` *`orderB`*`, const void *`*`B`*`, int` *`dC`*`, beanpole_order` *`orderC`*`, void *`*`C`*`)` [Function]

Do `C += A * B`, where A, B and C are polynomials in n variables of degree dA, dB and dC, respectively, with coefficients in the respective orders.

It is an error to invoke this function if the degree of C is less than that of the product of A and B, in other terms if $dC < dA + dB$.

The running time of this function is less than dC times the size of A times the size of B.

`void beanpole_product_mxp(beanpole_addmul_fn` *`updater`*`, void *`*`ctx`*`, int` *`n`*`, void *`*`L`*`, size_t` *`i`*`, int` *`k`*`, const int` *`m`*`[], int` *`dA`*`, beanpole_order` *`orderA`*`, const void *`*`A`*`, int` *`dB`*`, beanpole_order` *`orderB`*`, void *`*`B`*`)` [Function]

Do `B += L[i] * m * A`, where m is a degree-k monomial, A is a degree-dA polynomial and B is a degree-dB polynomial (both in n variables).

It is an error to call this function if the degree of the product is larger than that of B (i.e. if $dB < k + dA$). This function runs in time proportional to dB times the size of A.

7 The FES Algorithm

The FES algorithm provides an efficient iterator over the truth table of a Boolean polynomial. The iteration modifies the coefficients of the polynomial (denoted by A) and maintains the current evaluation point (denoted by x).

The following invariant is maintained at all times during the iteration:

> `A[0]` contains the result of the evaluation of the polynomial on x.

Values of x are enumerated using the order defined by the binary reflected Gray code. This implementation is almost in-place: it modifies the polynomial, but it is possible to restore its original state after the fact.

`void beanpole_fes_prepare(beanpole_add_fn *updater*, void *ctx*, int *n*, int *d*, void *A*, bool *x*[], struct beanpole_iterator *it*)` [Function]

Prepare A and sets x to $(0, 0, ..., 0)$.

The array x must be preallocated of size $n + 1$. The running time of this functions is upper-bounded by $d^2 phi^d$ times the size of A, where $phi = (1 + sqrt(5))/2$ is the golden ratio.

`void beanpole_fes_advance(void *A, bool x[], struct beanpole_iterator *it)` [Function]

Advance x to the next n-bit string and update A accordingly.

This functions executes $O(d)$ instructions.

`bool beanpole_fes_finished(const bool x[], const struct beanpole_iterator *it)` [Function]

Return true when all entries of the truth table have been visited.

This happens after 2^n iterations. This function runs in constant time.

`void beanpole_fes_restore(void *A, struct beanpole_iterator *it)` [Function]

Restore A to its initial value — before the call to `beanpole_fes_prepare`.

This assumes that the iteration is finished (`beanpole_fes_finished` has returned `true`).

The running time is about the same as that of `beanpole_fes_prepare`.

The demonstration programs provide several uses of these functions. Here is a quick, generic loop through the truth table:

```
/* iterate through the truth table */
bool x[n + 1];
struct beanpole_iterator it;
beanpole_fes_prepare(f, ctx, n, d, A, x, &it);
while (!beanpole_fes_finished(x, &it)) {
        /* do something with A[0] == f(x) */
        beanpole_fes_advance(A, x, &it);
}
beanpole_fes_restore(A, &it);
```

8 The in-place Moebius Transform

The in-place Moebius transform produces the truth table of a degree-d polynomial in successive chunks of size 2^d. It is an error to use it with $n < d$.

At all times during the iteration, the following invariant holds:

> The first 2^d entries of A contains the truth table of f where
> the last $(n - d)$ variables are fixed to `x[d:n]`.

`void beanpole_moebius_prepare(beanpole_add_fn` *`updater`*`, void *`*`ctx`*`, int `*`n`*`, int `*`d`*`, void *A, bool x[], struct beanpole_iterator *`*`it`*`)` [Function]

Prepare A and sets x to $(0, 0, ..., 0)$.

x must have size at least $n + 1$.

The first chunk of the truth table is available in A on exit.

This functions runs in time $O(d2^d)$.

`void beanpole_moebius_advance(void *A, bool x[], struct beanpole_iterator *`*`it`*`)` [Function]

Advance x[d:n] to the next value in lexicographic order, and update A accordingly.

All of A may be modified. The first d coefficients of x are not accessed.

This functions runs in amortized time $O(d2^d)$.

`bool beanpole_moebius_finished(const bool x[], const struct beanpole_iterator *`*`it`*`)` [Function]

Return `true` when the whole truth table has been visited.

In this case, A has been returned to its original state. This happens after 2^{n-d} calls to `beanpole_moebius_advance`.

This function runs in constant time.