LibTomPoly User Manual
v0.04

Tom St Denis
tomstdenis@iahu.ca

May 5, 2004

Open Source. Open Academia. Open Minds.

Tom St Denis,
Ontario, Canada

Contents

List of Figures

Chapter 1

Introduction

1.1 What is LibTomPoly?

LibTomPoly is a public domain open source library to provide polynomial basis arithmetic. It uses the public domain library LibTomMath (not included) for the integer arithmetic and extends the functonality to provide polynomial arithmetic.

Technically speaking the library allows the user to perform arithmetic on elements from the group $GF(p)[x]$ and to a lesser extent (this will change in the future) over $\mathbb{Z}[x]$. Essentially the math you can do with integers (including forming rings and fields) you can do with with polynomials and now you can do with LibTomPoly.

1.2 License

LibTomPoly is public domain. Enjoy.

1.3 Terminology

Throughout this manual and within the library there will be some terminology that not everyone is familiar with. It is afterall weird math.

Term	**Definition**
monic polynomial	A polynomial where the leading coefficient is a one.
irreducible polynomial	A polynomial that has no factors in a given group. For instance, $x^2 + 4$ is irreducible in $\mathbb{Z}[x]$ but not in $GF(17)[x]$ since $x^2 + 4 = (x + 8)(x + 9) \pmod{17}$.
primitive polynomial	An irreducible polynomial which generates all of elements of a given field (e.g. $GF(p)[x]/v(x)$)
characteristic	An integer k such that $k \cdot p(x) \equiv 0$
deg()	Functon returns degree of polynomial, e.g. $\deg(x^6 + x^3 + 1) = 6$

Figure 1.1: Terminology

1.4 Building the Library

The library is not ready for production yet but you can test out the library manually if you want. To build the library simply type

```
make
```

Which will build "libtompoly.a". To build a Win32 library with MSVC type

```
nmake -f makefile.msvc
```

To build against this library include "tompoly.h" and link against "libtompoly.a" (or tommath.lib as appropriate). To build the included demo type

```
make demo
```

Which will build "demo" in the current directory. The demo is not interactive and produces results which must be manually inspected.

Chapter 2

Getting Started

2.1 The LibTomMath Connection

LibTomPoly is really just an extension of LibTomMath[1]. As such the library has been designed in much the same way as far as argument passing and error handling events are concerned. The reader is encouraged to become familiar with LibTomMath before diving into LibTomPoly.

2.2 The pb_poly structure

A polynomial can characterized by a few variables. Given the C structure as follows

```
typedef struct {
   int    used,                 /* number of terms */
          alloc;                /* number of terms available (total) */
   mp_int characteristic,       /* characteristic, zero if not finite */
          *terms;               /* terms of polynomial */
} pb_poly;
```

1. The **used** member indicates how many terms of the **terms** array are used to represent the polynomial.

[1] http://math.libtomcrypt.org

2. The **alloc** member indicates the size of the **terms** array. Also note that even if **used** is less than **alloc** the mp_ints above **used** in the array must be set to a valid representation of zero.

3. The **characteristic** member is an mp_int representing the characteristic of the polynomial. If the desire is to have a null characteristic (e.g. $\mathbb{Z}[x]$) this element must still be initialized to a valid representation of zero.

4. The **terms** member is a dynamically sized array of mp_int values which represent the coefficients for the terms of the polynomial. They start from least to most significant degree. E.g. $p(x) = \sum_{i=0}^{used-1} terms_i \cdot x^i$.

2.3 Return Codes

The library uses the return codes from LibTomMath. They are

Code	**Meaning**
MP_OKAY	The function succeeded.
MP_VAL	The function input was invalid.
MP_MEM	Heap memory exhausted.
MP_YES	Response is yes.
MP_NO	Response is no.

Figure 2.1: Return Codes

2.4 Function Argument Passing

Just like LibTomMath the arguments are meant to be read left to right where the destination is on the right. Consider the following.

```
pb_add(a, b, c);   /* c = a + b */
pb_mul(a, b, c);   /* c = a * b */
```

Also like LibTomMath input arguments can be specified as output arguments. Consider.

```
pb_mul(a, b, a);   /* a = a * b */
pb_gcd(a, b, b);   /* b = (a, b) */
```

However, polynomial math raises another consideration. The characteristic of the result is taken from the right most argument passed to the function. Not all functions will return an error code if the characteristics of the inputs do not match so it's important to keep this in mind. In general the results are undefined if not all of the polynomials have identical characteristics.

2.5 Initializing Polynomials

In order to use a pb_poly structure with one of the functions in this library the structure must be initialized. There are three functions provided to initialize pb_poly structures.

2.5.1 Default Initialization

```
int pb_init(pb_poly *a, mp_int *characteristic);
```

This will initialize "a" with the given "characteristic" such that the polynomial represented is a constant zero. The mp_int characteristic must be a valid initialized mp_int even if a characteristic of zero is desired. By default, the polynomial will be initialized so there are "PB_TERMS" terms available. This will grow automatically as required by the other functions.

2.5.2 Initilization of Given Size

```
int pb_init_size(pb_poly *a, mp_int *characteristic, int size);
```

This behaves similar to pb_init() except it will allocate "size" terms to initialize instead of "PB_TERMS". This is useful if you happen to know in advance how many terms you want.

2.5.3 Initilization of a Copy

```
int pb_init_copy(pb_poly *a, pb_poly *b);
```

This will initialize "a" so it is a verbatim copy of "b". It will copy the characteristic and all of the terms from "b" into "a".

2.5.4 Freeing a Polynomial

```
int pb_clear(pb_poly *a);
```

This will free all the memory required by "a" and mark it as been freed. You can repeatedly pb_clear() the same pb_poly safely.

Chapter 3

Basic Operations

3.1 Comparison

Comparisions with polynomials is a bit less intuitive then with integers. Is x^2+3 greater than x^2+x+4? To create a rational form of comparison the following comparison codes were designed.

Code	**Meaning**
PB_EQ	The polynomials are exactly equal
PB_DEG_LT	The left polynomial has a lower degree than the right.
PB_DEG_EQ	Both have the same degree.
PB_DEG_GT	The left polynomial has a higher degree than the right.

Figure 3.1: Compare Codes

```
int pb_cmp(pb_poly *a, pb_poly *b);
```

This will compare the polynomial "a" to the left of the polynomial "b". It will return one of the four codes listed above. Note that the function does not compare the characteristics. So if $a \in GF(17)[x]$ and $b \in GF(11)[x]$ were both equal to x^2+3 they would compare to PB_EQ. Whereas x^3+4 would compare to PB_DEG_LT, x^1+7 would compare to PB_DEG_GT and x^2+7 would compare to PB_DEG_EQ[1].

[1]If the polynomial a were on the left for all three cases.

3.2 Copying and Swapping

```
int pb_copy(pb_poly *src, pb_poly *dest);
```

This will copy the polynomial from “src” to “dest” verbatim.

```
int pb_exch(pb_poly *a, pb_poly *b);
```

This will exchange the contents of “a” with “b”.

3.3 Multiplying and Dividing by x

```
int pb_lshd(pb_poly *a, int i);
int pb_rshd(pb_poly *a, int i);
```

These will multiply (or divide, respectfully) the polynomial “a” by x^i. If $i \leq 0$ the functions return without performing any operation. For example,

```
pb_lshd(a, 2);  /* a(x) = a(x) * x^2 */
pb_rshd(a, 7);  /* a(x) = a(x) / x^7 */
```

Chapter 4

Basic Arithmetic

4.1 Addition, Subtraction and Multiplication

```
int pb_add(pb_poly *a, pb_poly *b, pb_poly *c);
int pb_sub(pb_poly *a, pb_poly *b, pb_poly *c);
int pb_mul(pb_poly *a, pb_poly *b, pb_poly *c);
```

These will add (subtract or multiply, respectfully) the polynomial "a" and polynomial "b" and store the result in polynomial "c". The characteristic from "c" is used to calculate the result. Note that the coefficients of "c" will always be positive provided the characteristic of "c" is greater than zero.

Quick examples of usage.

```
pb_add(a, b, c);  /* c = a + b */
pb_sub(b, a, c);  /* c = b - a */
pb_mul(c, a, a);  /* a = c * a */
```

4.2 Division

```
int pb_div(pb_poly *a, pb_poly *b, pb_poly *c, pb_poly *d);
```

This will divide the polynomial "a" by "b" and store the quotient in "c" and remainder in "d". That is

$$b(x) \cdot c(x) + d(x) = a(x) \tag{4.1}$$

The value of $deg(d(x))$ is always less than $deg(b(x))$. Either of "c" and "d" can be set to **NULL** to signify their value is not desired. This is useful if you only want the quotient or remainder but not both.

Since one of the destinations can be **NULL** the characteristic of the result is taken from "b". The function will return an error if the characteristic of "a" differs from that of "b".

This function is defined for polynomials in $GF(p)[x]$ only. A routine pb_zdiv()[1] allows the division of polynomials in $\mathbb{Z}[x]$.

4.3 Modular Functions

```
int pb_addmod(pb_poly *a, pb_poly *b, pb_poly *c, pb_poly *d);
int pb_submod(pb_poly *a, pb_poly *b, pb_poly *c, pb_poly *d);
int pb_mulmod(pb_poly *a, pb_poly *b, pb_poly *c, pb_poly *d);
```

These add (subtract or multiply respectfully) the polynomial "a" and the polynomial "b" modulo the polynomial "c" and store the result in the polynomial "d".

[1] To be written!

Chapter 5

Algebraic Functions

5.1 Monic Reductions

```
int pb_monic(pb_poly *a, pb_poly *b)
```

Makes "b" the monic representation of "a" by ensuring the most significant coefficient is one. Only defined over $GF(p)[x]$. Note that this is not a straight copy to "b" so you must ensure the characteristic of the two are equal before you call the function[1]. Monic polynomials are related to their original polynomial through an integer k as follows

$$a(x) \cdot k^{-1} \equiv b(x) \tag{5.1}$$

5.2 Extended Euclidean Algorithm

```
int pb_exteuclid(pb_poly *a, pb_poly *b,
                 pb_poly *U1, pb_poly *U2, pb_poly *U3);
```

This will compute the Euclidean algorithm and find values "U1", "U2", "U3" such that

$$a(x) \cdot U1(x) + b(x) \cdot U2(x) = U3(x) \tag{5.2}$$

[1]Note that $a == b$ is acceptable as well.

The value of "U3" is reduced to a monic polynomial. The three destination variables are all optional and can be specified as **NULL** if they are not desired.

5.3 Greatest Common Divisor

```
int pb_gcd(pb_poly *a, pb_poly *b, pb_poly *c);
```

This finds the monic greatest common divisor of the two polynomials "a" and "b" and store the result in "c". The operation is only defined over $GF(p)[x]$.

5.4 Modular Inverse

```
int pb_invmod(pb_poly *a, pb_poly *b, pb_poly *c);
```

This finds the modular inverse of "a" modulo "b" and stores the result in "c". The operation is only defined over $GF(p)[x]$. If the operation succeed then the following congruency should hold true.

$$a(x)c(x) \equiv 1 \pmod{b(x)} \tag{5.3}$$

5.5 Modular Exponentiation

```
int pb_exptmod (pb_poly * G, mp_int * X, pb_poly * P, pb_poly * Y);
```

This raise "G" to the power of "X" modulo "P" and stores the result in "Y". Or as a congruence

$$Y(x) \equiv G(x)^X \pmod{P(x)} \tag{5.4}$$

Where "X" can be negative[2] or positive. This function is only defined over $GF(p)[x]$.

5.6 Irreducibility Testing

```
int pb_isirreduc(pb_poly *a, int *res);
```

Sets "res" to MP_YES if "a" is irreducible (only for $GF(p)[x]$) otherwise sets "res" to MP_NO.

[2] But in that case $G^{-1}(x)$ must exist modulo $P(x)$.

Index