Galois Field Library: Reference Manual

Abhijit Das and C. E. Veni Madhavan
Department of Computer Science and Automation
Indian Institute of Science
Bangalore 560 012 INDIA
abhij,cevm@csa.iisc.ernet.in

TR No. IISc-CSA-98-05
February 1998

Abstract

Galois Field Library (GFL) is a portable general-purpose computational library of functions written in C for working over finite fields. The library provides a comprehensive treatment of operations in prime fields and their arbitrary finite extensions. Application programmers should find this library useful for developing programs in the areas of public-key cryptography, error control coding and combinatorial design.

This technical report is a reference manual of GFL. It provides an exhaustive listing of all the features of the library – namely the new data structures and macro definitions introduced in the header files of the library and the prototypes of all GFL library calls.

KEY WORDS: Finite fields Data structures Algorithms Library

1 Introduction

Galois Field Library (GFL) is a portable general-purpose computational library of functions written in C for working over finite fields (also called *Galois fields*). GFL provides routines for field arithmetic and for manipulation of univariate polynomials and matrices over finite fields. This library is, to the authors' knowledge, the first extensive implementation of arithmetic over arbitrary finite fields. The usefulness of this library stems from the following considerations.

1. *Generality:* GFL works on finite fields of any characteristic and any cardinality. It allows one to work both on prime fields and on their finite algebraic extensions obtained by adjoining an arbitrary number of (algebraic) elements. That is, one first creates prime fields and then defines extensions of these prime fields, extensions of these extensions, and so on. Previous implementations known to the authors can handle only specific types of fields (fields of characteristic 2, or prime fields).

2. *Extensiveness:* GFL provides extensive tools for a wide range of problems that are of computational importance in the theory of finite fields. These features include: computing minimal and

irreducible polynomials, finding primitive roots and normal elements, finding roots and factors of polynomials, solving linear systems of equations, computing isomorphism between two fields of the same cardinality, computing discrete logarithms, checking permutation polynomials, working with affine multiples.

3. *Performance:* Use of suitable data structures, fine tuning of basic arithmetic operations, and use of several implementation tricks such as table look-up and modularity make GFL a fast, memory-efficient and easy-to-use tool.

4. *Portability:* GFL can be used on any workstation that has an ANSI C compiler. It is totally self-contained in the sense that it is not built as a library over existing packages.

In what follows, we describe the conventions and features provided by GFL. We provide an exhaustive listing of all data structures, macro definitions and prototypes of library calls that pertain to GFL. We refer the reader to [8] for a short tutorial on the basic programming paradigms that the users should follow when they use GFL routines in their programs. While that tutorial is a nice and readable introduction to GFL, it is by no means a complete reference to GFL. This technical report is meant to be a supplement to [8].

GFL comes in two flavors. The one we describe in the first few sections of this report can be used for working over fields whose cardinalities fit in single computer words (data type `long int` in C). A multi-precision integer library is designed to cope up with larger fields. The other form of GFL that deals with these multi-precision numbers, will be termed the multi-precision Galois field library.

Randomized algorithms play a very important role in computations over finite fields. This is because for many of the common problems, deterministic polynomial time algorithms are not known. (By a polynomial time algorithm, we mean one that runs in time polynomially bounded by the logarithm of the cardinality of the field.) In many other cases, even if deterministic algorithms are known, they cannot normally compete in speed with their probabilistic counterparts. GFL implements the probabilistic versions of the algorithms whenever applicable or useful.

Finite field algorithms find immense applications in the areas of public-key cryptography, error control coding, combinatorial designs and so on. We expect that this library would be useful to programmers who develop application packages in these areas. We plan to distribute GFL as a *freeware* for academic and research purposes.

The rest of the report is organized as follows. In the next section we discuss about the general conventions and notations that we adopt in this report. In Sections 3 through 5 we explain how GFL represents the basic algebraic entities like finite fields, elements of finite fields, univariate polynomials, vectors and matrices over these fields. We then talk about the polynomial utilities implemented in GFL in Sections 6 through 11. These include computation of minimal and irreducible polynomials, finding roots and factorization of polynomials, computing least affine multiples of polynomials, checking permutation polynomials and so on. A description of the field functions comes next (Sections 12 through 16). These functions can be used for the computation of normal and primitive elements, transformation matrices between different bases, isomorphism between two fields of the same cardinality, discrete logarithms with respect to primitive elements etc. All these routines apply to the single-precision library. We then define multi-precision integers in Section 17 and explain in Section 18 how the prototypes of the single-precision library calls change in the multi-precision library. In Section 19, we illustrate the use of the library with the help of some small examples. Finally, we describe some debugging tools provided by GFL in Section 20. We mention some possible enhancements of GFL and conclude the paper in Section 21.

2 Conventions and notations

In what follows, we will refrain from defining algebraic terms and concepts that are well-known and can be found in text books on algebra [11], linear algebra [12] or finite fields [16, 18, 19, 23]. We define and/or explain terms that we introduce during the course of the report, i.e. those that are particular to GFL.

Some general conventions and notations that we follow in this report are:

1. Variable names are written in the sans serif font, whereas the *values* they represent (which are typically algebraic entities) are printed using the math-italic font. Thus, for example, if M is a variable of type matrix, we denote by M the value of M, i.e. the matrix represented by M.

2. The finite field of cardinality q is denoted as $\mathbb{F}_q$. Unless otherwise specified we will use the letter p to denote the characteristic of $\mathbb{F}_q$.

3. The indeterminate in a univariate polynomial is assumed to be x, that is, when we are talking about a variable f of type poly, we denote the value of f by $f(x)$ or simply by f, when no confusions are likely.

4. The ideal generated by a polynomial $f(x)$ is expressed as $<f>$ or $<f(x)>$. This notation does not reveal the ring in which $<f>$ is an ideal. Typically this is the ring of univariate polynomials over a given finite field.

5. We shall generally use the lower-case letters m, n, r, s, t to represent integers, the letters a, b, c etc. to stand for elements of finite fields, the letters f, g, h to represent polynomials, u, v, w to represent vectors, the upper-case letters F, K, L to denote finite fields and the letters A, B, M etc. to denote matrices.

3 Representation of finite fields

GFL maintains a *field descriptor* for every finite field created. This descriptor is of the data type GF_d (which is a short int). All references to the fields created can be done through these descriptors. GFL allows one to work with at most 64 field descriptors. This number may, however, be changed by changing MAX_FIELDS as defined in field.h and recompiling GFL. In addition to the descriptor, GFL maintains a set of information for every field created. These information are available by library calls with the field descriptor as argument. We will come to this later. Let us first see how one can create finite fields of prime and prime power cardinality.

3.1 Prime fields

A finite field of prime cardinality p is represented as an algebraic system where all arithmetic operations are integer operations modulo p. A prime field can be created by a call to the routine createPrimeGF that takes one argument, the characteristic of the field. It outputs a field descriptor for the field. createPrimeGF checks whether the argument supplied is a prime. If not, it outputs an error message and returns an invalid field descriptor.

```
#include <gfl/field.h>

GF_d F;
```

```
long p;

/* Assign a prime value to p */
p = 1009;

/* Create the prime field */
F = createPrimeGF(p);
```

3.2 Prime power fields

Unlike prime fields, a finite field of cardinality $q = p^s$ for $s > 1$ cannot be *created* as a stand-alone field. It has to be defined as an algebraic extension of an existing field (which might be a prime field or another non–prime field that has been already defined). The library routine createExtGF is to be used for this purpose. This routine takes two arguments. The first one is the descriptor of the field of which the new field is an extension. The second argument is a *monic irreducible* (over the field we are extending) polynomial of the desired degree. As in the call of createPrimeGF, a GF_d is returned by createExtGF for future references. Note that createExtGF demands the extending polynomial to be monic but does not check whether the polynomial is irreducible. If a non-irreducible polynomial is supplied, the resulting algebraic system is not a field (it has nonzero zero divisors) and the field arithmetic is no longer applicable to this GF_d. We will later discuss how one can work with polynomials (in particular, irreducible polynomials). Here let us assume that polynomials are represented as data of type poly.

```
#include <gfl/field.h>

GF_d F, K;
long p = 7;
poly f;

/* Create the prime field */
F = createPrimeGF(p);

/* Define f to be a suitable irreducible polynomial over F */
/* Assign the coefficients of f */

/* Create the extension field */
K = createExtGF(F,f);
```

Now K may be extended by an irreducible polynomial over K to get another field, say L, following the same procedure as above. In this example we call K to be an *immediate extension* of F and L an *immediate extension* of K.

```
GF_d L;
poly g;

/* Assign g to be an irreducible polynomial over K */
```

L = createExtGF(K,g);

L can be extended further thus giving an arbitrarily long[1] tower of fields. In addition, one may work with more than one (immediate) extension of the same field. That is, F can be simultaneously extended by a polynomial different from f, K by one different from g, and so on. It is also admissible to define more than one prime field. In short, GFL allows one to work with an arbitrary *directed forest* of fields. Each call of createExtGF (and also of createPrimeGF) returns a brand new GF_d as long as the field descriptor table is not full.

In short, GFL uses the vector–space representation of field extensions. Each extension is defined by the *polynomial basis*. Arithmetic in the extension is performed as polynomial arithmetic in the sub-field modulo the irreducible polynomial used for defining the field extension.

3.3 Information on finite fields

In addition to creating a field descriptor, a call to createPrimeGF or createExtGF creates an entry in an *information array* named GF_info for the field created. This array is indexed by the field descriptor returned. One may access these information either by accessing records of GF_info or by specific library calls. We list these in the following table. Here K and F are field descriptors (of data type GF_d). An empty entry in the first column implies that there is no equivalent library call, and the same in the second column implies that the corresponding piece of data is not maintained in the GF_info records.

Library call	**Explicit access**	**Data type**	**Meaning**
characteristic(K)	GF_info[K].p	GFelement	The characteristic of K
cardinality(K)	GF_info[K].q	GFelement	The cardinality of K
subGF(K)	GF_info[K].extof	GF_d	The field over which K is represented as an extension. This is undefined (-1) for a prime field created by createPrimeGF.
primeSubGF(K)		GF_d	The prime subfield of K.
extDeg(K,F)		int	The extension degree of K over F. A negative value is returned, if K is not represented as an extension (or extension of extension or ...) of F.
	GF_info[K].extdeg	int	The extension degree of K over F, if K is defined as an algebraic extension of F. This is the same as the degree of f if K is defined as $F/<f>$. This is undefined (0) if K is a prime field.
totExtDeg(K)	GF_info[K].totextdeg	int	The extension degree of K over its prime subfield.
extLevel(K)	GF_info[K].extlev	int	Level of extension of K over its prime subfield, that is, the number of elements adjoined to the prime subfield to get K.

[1]The number of fields is limited by MAX_FIELDS as already described.

defPoly(K,&f)		void	The defining polynomial of K over its *immediate subfield* is copied to f. If K is the prime field, this is undefined.
	GF_info[K].defpoly	GFelement*	Pointer to the coefficient array of f, the defining polynomial of K over its *immediate subfield*. NULL for prime fields.
printGFInfo(K)		void	Print the above information (and some more that we will discuss later) on K.

We do not explain here all the fields of the GF_info records. See Section 13 to have a more complete definition of these structures.

3.4 Representation of finite field elements

Elements of a finite field $\mathbb{F}_q$ of cardinality q are represented as integers between 0 and $q-1$. For fields of prime cardinality, this is an obvious representation. For fields of prime power cardinality, this has the following interpretation. First let us assume that $\mathbb{F}_q$ is an extension of $\mathbb{F}_p$ defined by a polynomial $f(x) \in \mathbb{F}_p[x]$ of degree s (so that $q = p^s$). Let α be a root of $f(x)$ in $\mathbb{F}_q[x]$. Then an element $c \in \mathbb{F}_q$ can be uniquely represented as $c = c_{s-1}\alpha^{s-1} + c_{s-2}\alpha^{s-2} + \cdots + c_1\alpha + c_0$, where $c_{s-1}, c_{s-2}, \cdots, c_1, c_0 \in \mathbb{F}_p$. We may view c as an s-digit number $c_{s-1}c_{s-2}\cdots c_1c_0$ in base p. Then c is an integer between 0 and $q-1$. Note that $1, \alpha, \alpha^2, \ldots, \alpha^{s-1}$ constitute the polynomial basis of $\mathbb{F}_q$ over $\mathbb{F}_p$. Viewed as numbers, these basis elements are respectively $1, p, p^2, \ldots, p^{s-1}$.

Next let us extend $\mathbb{F}_q$ by $g(x) \in \mathbb{F}_q[x]$ of degree t to get the field $\mathbb{F}_{q^t} = \mathbb{F}_q[x]/<g>$. If β is a root of $g(x)$ in $\mathbb{F}_{q^t}$, then an element $c \in \mathbb{F}_{q^t}$ has the unique representation $c = c_{t-1}\beta^{t-1} + c_{t-2}\beta^{t-2} + \cdots + c_1\beta + c_0$, where $c_{t-1}, c_{t-2}, \cdots, c_1, c_0 \in \mathbb{F}_q$. We may, therefore, represent c as the t-digit number $c_{t-1}c_{t-2}\cdots c_1c_0$ in base $q = p^s$. Each c_i, on the other hand, can be represented as an s-digit number $c_{i,s-1}c_{i,s-2}\cdots c_{i,1}c_{i,0}$ in base p, so that

$$
\begin{aligned}
c \;=\;& (c_{t-1,s-1}\alpha^{s-1} + c_{t-1,s-2}\alpha^{s-2} + \cdots + c_{t-1,1}\alpha + c_{t-1,0})\beta^{t-1} \\
&+ (c_{t-2,s-1}\alpha^{s-1} + c_{t-2,s-2}\alpha^{s-2} + \cdots + c_{t-2,1}\alpha + c_{t-2,0})\beta^{t-2} \\
&+ \cdots \\
&+ (c_{1,s-1}\alpha^{s-1} + c_{1,s-2}\alpha^{s-2} + \cdots + c_{1,1}\alpha + c_{1,0})\beta \\
&+ (c_{0,s-1}\alpha^{s-1} + c_{0,s-2}\alpha^{s-2} + \cdots + c_{0,1}\alpha + c_{0,0})
\end{aligned}
$$

Hence we may view c also as the st-digit number

$$c_{t-1,s-1}c_{t-1,s-2}\cdots c_{t-1,1}c_{t-1,0}c_{t-2,s-1}c_{t-2,s-2}\cdots c_{t-2,1}c_{t-1,0}\cdots c_{0,s-1}c_{0,s-2}\cdots c_{0,1}c_{0,0}$$

in base p. Therefore c is a number between 0 and $p^{st}-1$. Here $1, \beta, \beta^2, \ldots, \beta^{t-1}$ form the polynomial basis of $\mathbb{F}_{q^t}$ over $\mathbb{F}_q$. GFL represents these basis elements as the numbers $1, q, q^2, \ldots, q^{t-1}$. Note also that the elements $1, \alpha, \ldots, \alpha^{s-1}, \beta, \beta\alpha, \ldots, \beta\alpha^{s-1}, \ldots, \beta^{t-1}, \beta^{t-1}\alpha, \ldots, \beta^{t-1}\alpha^{s-1}$ is a basis of $\mathbb{F}_{p^{st}}$ over $\mathbb{F}_p$. This is not a polynomial basis. We call it a *composed basis* of $\mathbb{F}_{p^{st}}$ over $\mathbb{F}_p$. In GFL, this basis is represented as the set of numbers $1, p, p^2, \ldots, p^{st-1}$.

It's clear that this representation of finite field elements by numbers can be similarly extended to extensions of $\mathbb{F}_{q^t}$, to extensions of these extensions, and so on. To sum up, for any q the elements of $\mathbb{F}_q$ are represented as integers between 0 and $q-1$ irrespective of the definition of $\mathbb{F}_q$. The interpretation of the numbers is, however, dependent on the definition of $\mathbb{F}_q$. We call this representation of finite field elements the *packed representation* as contrasted with the *unpacked representation* in which elements are represented as tuples or polynomials. It should be worthwhile to discuss the relative merits and demerits of these two representations before we proceed further.

- The packed representation needs less memory than its unpacked counterpart.
- The packed representation is a uniform representation of a finite field element irrespective of the field to which the element belongs. This means that if an extension K of F is defined where $\mathrm{card}(F) = q$, an element $c \in K$ with $0 \leq c \leq q-1$ is automatically an element of F and has the *same* interpretation in both K and F. For example, the number 0 (resp. 1) represents the additive (resp. multiplicative) identity in any field. There is no overhead of typecasting elements of one field to those of another. This saves time.
- Indexing arrays etc. by finite field elements or letting a loop variable run over finite field elements become easier with the representation of the elements as integers. This too speeds up arithmetic.
- Almost all arithmetic operations over finite fields require the individual elements of the unpacked representation. This means that for every such operation the operands should first be unpacked. After the operation the result should be packed and returned. This adds to the cost of arithmetic. This overhead is negligible during computation of products and powers, whereas for sums and differences, we cannot neglect the effect of packing and unpacking. GFL is designed to keep this overhead at a bare minimum.
- When we are working over fields of characteristic 2, the individual bits of the unpacked representation remain "visible" in the sense that packing and unpacking can be done by bit operations only, which are very fast. In addition, the procedure that implements sum (and difference) over these fields need not separate the individual bits of the operands. A single XOR operation on a full word can process all the bits in the word simultaneously.

Back to programming issues. The header file field.h defines the data type GFelement to represent an element of a finite field. This is essentially a long int. A variable of type GFelement can be assigned a value as

```
#include <gfl/field.h>

GFelement c;

c = 576;
```

One should take care that when c is treated as an element of a field of cardinality q, one must have $0 \leq c \leq q-1$. One may expect bizarre results when inadmissible values of c are passed to the built-in library routines. Most of these routines do not check for the correctness of values passed to them. Some of them implicitly or explicitly replaces them by their remainders on division by the cardinality of the appropriate field.

Elements of a finite field K can be printed by the routine printGFElement that takes four arguments – the number (of type GFelement), the field descriptor for interpretation of this number, a file pointer (FILE *) opened in the write or append mode to redirect the output, and a flag. printGFElement returns the number of characters printed. If the special value NULL is supplied as the file pointer, the output is redirected to stdout. The flag indicates how to format the output.

```
printGFElement(a, K, stdout, 2);
```

As an example, we print in the following table an element of $\mathbb{F}_{5^6}$ that has been defined by extending $\mathbb{F}_5$ first with a polynomial of degree 2 and then extending this intermediate field by another polynomial of degree 3.

flag	Interpretation of flag	Example
0	Single number	12345
1	Vector over the immediate subfield of K	(19 18 20)
2	Vector of vectors of ... over prime subfield	((3 4) (3 3) (4 0))
3	Flattened form of 2	(3 4 3 3 4 0)
4	Same as 3 but without parentheses	3 4 3 3 4 0
5	Polynomial in last extending element	(19b^2 + 18b + 20)
anything else	Polynomial in extending elements	((3a + 4)b^2 + (3a + 3)b + (4a + 0))

3.5 Routines for doing arithmetic over finite fields

Let us summarize the basic arithmetic operations provided by GFL. In the following table, K denotes a field descriptor (of type GF_d), a, b, e are GFelement's. Each function returns a GFelement which should be interpreted as an element of K.

Routine	Return value
GFsum(a,b,K)	$a + b$
GFdiff(a,b,K)	$a - b$
GFprod(a,b,K)	ab
GFqt(a,b,K)	$\frac{a}{b},\ b \neq 0$
GFinv(a,K)	$a^{-1},\ a \neq 0$
GFexp(a,e,K)	a^e

In addition to the above basic routines, GFL provides the following useful functions. K is assumed to be an extension (not necessarily immediate) of F, with $\mathrm{card}(F) = q$ and $\mathrm{card}(K) = q^s$. If K is not an extension of F, each of the functions returns an undefined GFelement (-1).

Routine	Return value
trace(a,K,F)	$\mathrm{tr}_{K\vert F}(a) = a + a^q + a^{q^2} + \cdots + a^{q^{s-1}}$
absTrace(a,K)	$\mathrm{tr}_{K\vert F}(a),$ where F is the prime subfield of K
norm(a,K,F)	$\mathrm{N}_{K\vert F}(a) = a \cdot a^q \cdot a^{q^2} \cdots a^{q^{s-1}}$
absNorm(a,K)	$\mathrm{N}_{K\vert F}(a),$ where F is the prime subfield of K

4 Polynomials over finite fields

Polynomials are at the very heart of representing finite field extensions. The coefficients of a polynomial are stored in a *dynamic* array. The advantage of this representation over the representation by static arrays is that in the former representation passing polynomials to subroutines is much faster than that in the latter. This is because with dynamic arrays only a pointer to the coefficient array need to be passed instead of the entire array. This also facilitates efficient use of memory since the pointers can be allocated only as much memory as is needed to hold the array. On the other hand, this representation makes programming a little difficult. The programmer has to be careful enough while allocating and freeing memory associated with a polynomial, in particular, inside his/her own subroutines. This leads to potential sources of bugs in programs that use GFL. In a latter section, we will discuss about these problems and their possible remedies. Before that, let's see how we can define, initialize, assign and do arithmetic with polynomials.

4.1 Representation of polynomials

All arithmetic routines for manipulation of polynomials are declared in the header file poly.h. However, the basic data structure for a polynomial is defined in the header file field.h. This is a structure of two elements. The first one is the *exact degree* of the polynomial and the second a pointer to the coefficient array of the polynomial. The coefficients are of data type GFelement. As in the case of GFelement, the definition of a polynomial does not mention a field to which the polynomial is intended to belong. When one calls a routine that does arithmetic on polynomials, one must specify the field (GF_d) over which the coefficient arithmetic should take place.

```
typedef struct {
    int degree;             /* The exact degree */
    GFelement *coeff;       /* Pointer to the array of coefficients */
} poly;
```

If f is a variable of data type poly and f.degree = $d \geq 0$, the coefficient of x^i of f can be accessed as f.coeff[i] for $0 \leq i \leq d$. In particular, f.coeff[f.degree] is the *leading coefficient* of f. The *zero polynomial* has the following representation

```
f.degree = MINUS_INFINITY;
f.coeff = NULL;
```

MINUS_INFINITY is a macro defined in poly.h that represents the degree of the zero polynomial (It stands for the value of -1).

It is strongly recommended to "initialize" a polynomial before it is used. This can be done by a call to the library routine newPoly().

```
#include <gfl/field.h>
#include <gfl/poly.h>
poly f;
f = newPoly();
```

newPoly returns the zero polynomial. Instead of calling newPoly, one can achieve the same as

```
f.degree = MINUS_INFINITY;
f.coeff = NULL;
```

Once a polynomial is initialized, it can be assigned values. One can do that in several ways. One can assign the individual struct components manually as in the following example.

```
int d = ...;

f.degree = d;
f.coeff = (GFelement *)malloc((d + 1) * sizeof(GFelement));
f.coeff[0] = ...;
f.coeff[1] = ...;
...
```

```
f.coeff[d] = ...;
```

This completes the process. There is only one thing one should remember. To avoid unpleasant situations, it is mandatory to assign f.coeff[f.degree] to a nonzero value. As pointed out earlier, the element degree of a poly structure should contain the *exact degree* of the polynomial.

Reassignment of a polynomial is a little bit involved. Let's take the last code example. If f is reassigned to a polynomial of same degree d, there is no problem. One can simply assign the coefficients the new values. Otherwise, it is advisable to do the following to make the programs memory-efficient and bug-free.

```
f.degree = d';
if (f.coeff != NULL) free(f.coeff);
    /* Return the memory currently allocated to f.coeff to the pool for future reuse */
f.coeff = NULL;                    /* This may not be necessary. But preferably do it. */
f.coeff = (GFelement *)malloc((d' + 1) * sizeof(GFelement));
                                   /* Reallocate memory to f.coeff */
/* Now set the coefficients */
f.coeff[0] = ...;
f.coeff[1] = ...;
...
f.coeff[d'] = ...;
```

The C routine realloc may also be used for reallocating memory, but this routine is inefficient compared to malloc in situations as in the last code snippet. This is because realloc copies memory contents from the old location to the new location. Another memory allocation routine is calloc. This is also slow compared to malloc, because a call of calloc initializes every location in the allocated memory to zero. calloc can, however, be used when such initializations are needed. See [14] for details.

GFL provides a library routine destroyPoly to free memory allocated to the coefficient array of a polynomial. It accepts a pointer to a polynomial as its sole argument, calls free on the coefficient array of the polynomial, makes this array NULL and sets the degree to MINUS_INFINITY. When destroyPoly returns, its argument is thus set to the zero polynomial.

```
destroyPoly(&f);
```

There are alternative ways to assign or reassign a polynomial. The procedure readPoly reads a polynomial interactively from stdin and the procedure readPolyFromArray reads a polynomial from an array of GFelement's. Both these routines adjust the degree of the assigned polynomial to the exact degree of the polynomial. One is allowed to declare the degree of the polynomial to be, say, d and supply 0 as coefficients of x^d, x^{d-1} and so on (till as far as x^0). The third routine we describe in this respect is copyPoly which copies a polynomial to another. Each of these three routines frees memory, if any, currently held by the polynomial being assigned. The user need not do it explicitly (though there is no harm in doing so).

```
#include <gfl/field.h>
#include <gfl/poly.h>

GF_d K;
```

```
poly f, g, h;
GFelement ca[5] = 1, 0, 3, 4, 2;

/* First initialize the polynomials */
f = newPoly();
g = newPoly();
h = newPoly();

/* read f interactively */
readPoly(&f);

/* read g from ca */
readPolyFromArray(&g,4,ca);

/* copy f or g to h */
if (f.degree > g.degree)
    copyPoly(&h, f);
else
    copyPoly(&h, g);
```

In this example, the polynomial $1 + 3x^2 + 4x^3 + 2x^4$ is assigned to g. Also note that copyPoly(&h, f) is not the same as the assignment h = f. The former makes a verbatim copy of the coefficients of f to the coefficients of h, whereas in the assignment h = f, the pointer to the coefficient array of f is copied to the pointer to the coefficient array of h. Any subsequent modification of the coefficient array of h also modifies that of f and vice versa. Such assignments should, therefore, be avoided. Finally, note that copyPoly does not allow one to copy a polynomial to itself. If it is requested to do so, it returns with a warning message.

We are left with only one routine for this section: writePoly for printing a polynomial. It takes four inputs: the polynomial to be printed, a field descriptor to imply the field over which the polynomial is defined, a file pointer (FILE *) opened for writing (or NULL to mean stdout) and a flag indicating how to format the coefficients. This flag has the same meaning as in printGFElement. writePoly returns the number of characters printed. Here we give an example. We want to print the polynomial $4567x^2 + 9876x + 12345$ over $\mathbb{F}_{5^6}$ that is defined as an extension of $\mathbb{F}_{5^2}$ which, in turn, is defined as an extension of the prime field $\mathbb{F}_5$. We tabulate the output of writePoly on this polynomial with different flag values. (New lines are sometimes introduced for clarity.)

flag	**output**
0	4567x^2 + 9876x + 12345
1	(7 7 17)x^2 + (15 20 1)x + (19 18 20)
2	((1 2) (1 2) (3 2))x^2 + ((3 0) (4 0) (0 1))x + ((3 4) (3 3) (4 0))
3	(1 2 1 2 3 2)x^2 + (3 0 4 0 0 1)x + (3 4 3 3 4 0)
4	1 2 1 2 3 2x^2 + 3 0 4 0 0 1x + 3 4 3 3 4 0
5	(7b^2 + 7b + 17)x^2 + (15b^2 + 20b + 1)x + (19b^2 + 18b + 20)
6	((1a + 2)b^2 + (1a + 2)b + (3a + 2))x^2 + ((3a + 0)b^2 + (4a + 0)b + (0a + 1))x + ((3a + 4)b^2 + (3a + 3)b + (4a + 0))

4.2 Arithmetic operations on polynomials

GFL provides built-in procedures for arithmetic operations in the ring $\mathbb{F}_q[x]$. A binary operation on polynomials has the general format

`op(&f, g, h, K);`

This means "apply the operation `op` on the polynomials `g` and `h` and save the result in the polynomial `f`, where the coefficient arithmetic is that of the field K", that is,

`f = g op h` (in $K[x]$)

These functions return `void`. It's admissible to supply the same polynomial both as an operand and as the output. That is, calls like

`op(&f, f, g, K);`

or even

`op(&f, f, f, K);`

are allowed.

The seemingly better alternative of returning a `poly` from a subroutine to which only the operands are passed has one drawback. Consider the following syntax:

`f = altop(g, h, K);`

This means that copies of `g` and `h` are passed to `altop` and the return value is stored `f`. In this case the memory allocated to `f` (if any) is not freed before it is assigned a newly allocated memory. The GFL implementation, on the other hand, takes care that the memory allocated to the coefficient array is freed before it is reallocated memory for holding the return value. It is also ensured that in a call like

`op(&f, f, g, K);`

the coefficient array of `f` is freed only when it is no longer needed as an operand.

Without further ado we summarize the built-in arithmetic routines in $\mathbb{F}_q[x]$ as provided by GFL. In the following tables, `f`, `g`, `h`, `m`, `u` and `v` are polynomials (of type `poly`) over $K[x]$, `K` is a field descriptor, `a` and `e` are `GFelement`'s.

Call	**Effect**
polySum(&f, g, h, K)	$f = g + h$
polyDiff(&f, g, h, K)	$f = g - h$
polyProd(&f, g, h, K)	$f = gh$
polyDiv(&f, &m, g, h, K)	$f = g/h,\ m = g \bmod h$
polyExp(&f, g, e, K)	$f = g^e$
polyModProd(&f, g, h, m, K)	$f = gh \bmod m$
polyModExp(&f, g, m, e, K)	$f = g^e \bmod m$
polyGCD(&f, g, h, K)	$f = \gcd(g, h)$
polyEGCD(&f, &u, &v, g, h, K)	$f = \gcd(g, h) = ug + vh$ (Extended GCD)
lc(f)	Return the leading coefficient of f
monic(f)	Check if f is monic (boolean)
monicize(&f)	$f = a^{-1}f$ where a is the leading coefficient of f
zeroPoly(f)	Check if f is the zero polynomial
equalPoly(f, g)	Check if f and g are equal
evalPoly(f, a, K)	Evaluate $f(x)$ at $x = a$ and return this value

Note that one is allowed to pass to `polyDiv` the `NULL` pointer as the first or the second argument (or both). That is, if one is interested in calculating only the remainder of a polynomial division, one can call

`polyDiv(NULL, &m, g, h, K)`

4.3 Potential bugs with dynamic arrays and their remedies

In a previous section, we mentioned the possibility of bugs that might creep in programs that use GFL. It's now time to concentrate on them. We restate that the second struct element of a poly is a pointer to the coefficient array of the polynomial rather than the array itself. This might create troubles unless proper care is taken. To see how, let's take an example. Suppose we are interested in writing a routine that outputs the monic square of a polynomial. Consider the following routine.

```
void monicSquare ( poly *f , poly g , GF_d K ) {
    /* First monicize g before squaring */
    monicize(&g);

    /* Then multiply g with itself */
    polyProd(&g, g, g, K);

    /* Now copy the product to (*f) */
    copyPoly(f, g);
}
```

Now consider the call

```
monicSquare(&ff, gg, K)
```

where ff and gg are previously defined polynomials. We intend to store the monic square of gg in ff. As we make the function call, gg gets copied in monicSquare with the name g. The first thing the subroutine does is monicizing g. If the leading coefficient of g is not equal to 1, this changes the coefficient array of g *in situ*. This automatically changes the coefficient array of gg also, because g.coeff and gg.coeff are two different pointers that point to the *same* memory location. To worsen matters, the subsequent call of polyProd frees the memory associated with g.coeff before assigning it the multiplication result. This means that now the run-time system is permitted to allocate to another data the memory which is *legally* held by gg.coeff. To avoid such unpleasant situations, it should be kept in mind: *One should never modify the coefficient array (implicitly or explicitly) of a polynomial passed as an argument to a subroutine. The array should first be copied to a temporary location. All subsequent modifications should be applied to this copy.* So a correct implementation of monicSquare can look like:

```
void monicSquare ( poly *f , poly g , GF_d K ) {
    int i;
    GFelement *ca = g.coeff;
        /* Let ca point to the memory location initially pointed to by g */

    /* Assign g.coeff a copy of the array pointed to by ca */
    g.coeff = NULL;
    g.coeff = (GFelement *)malloc((g.degree + 1) * sizeof(GFelement));
    for (i=0; i<=g.degree; i++) g.coeff[i] = ca[i];

    /* Now we have prepared ground to modify g indiscriminately */
    monicize(&g);
    polyProd(&g, g, g, K);
```

```
    copyPoly(f, g);

    /* If memory is a scarce resource, the following is recommended */
    free(g.coeff);
}
```

Too much for a simple routine!!! May be. As we claimed earlier, one of the reasons for introducing pointers to replace arrays was to avoid making copies of the entire array on every function call. And here we have to do the copying task ourselves. True. But every function does not need to modify all of its polynomial arguments and it is there where we enjoy the advantage of pointers. In addition the following trick makes the function look simpler and run more efficiently.

```
void monicSquare ( poly *f , poly g , GF_d K ) {
    /* First copy g to f unless they refer to the same polynomial */
    if ((*f).coeff != g.coeff)
        copyPoly(f,g);

    /* Change f instead of g */
    monicize(f);
    polyProd(f,*f,*f,K);
}
```

The if condition was necessary if we want to avoid the warning message

```
WARNING in copyPoly:: attempt to copy polynomial to itself
```

Of course, the copyPoly operation is unnecessary if ((*f).coeff == g.coeff). f can be changed at will, because it is meant for getting modified by the function call. But there is a point to remember here. One must not change f, if that change has some effect on g. This occurs when *f and g refer to the same polynomial as in the call

```
monicSquare(&ff, ff, K);
```

So any change one makes in (*f).coeff is reflected in g.coeff as well and there is no way to recover the original coefficient array of g. This is another potential source of bug and can be avoided by modifying a temporary polynomial and assigning it to f when coefficients of g are no longer needed. This is exemplified by the following routine that does assignment of the form $f = g^2 + g$. First we write the faulty routine.

```
void faultySquarePlus( poly *f , poly g , GF_d K ) {
    if ((*f).coeff != g.coeff) copyPoly(f, g);

    polyProd(f, *f, *f, K);
    polySum(f, *f, g, K);
}
```

If ff and gg are two different polynomials and we call

```
faultySquarePlus(&ff, gg, K);
```

it correctly assigns $gg^2 + gg$ to ff. But if we call

```
faultySquarePlus(&ff, ff, K);
```

we may expect bizarre results, because polyProd frees the memory pointed to by (*f).coeff which is the same as that by g.coeff. This memory might be reallocated to some other data (for example, (*f).coeff itself) before polySum is called, so that g.coeff looses its identity. A temporary polynomial is necessary in this case as in the following corrected routine.

```
void squarePlus( poly *f , poly g , GF_d K ) {
    poly temp;

    temp = newPoly();       /* Initialize temp */
    copyPoly(&temp, g);
    polyProd(&temp, temp, temp, K);
    polySum(f, temp, g, K);       /* Coefficients of g are not needed after this step */

    /* Don't forget to free temporary memory */
    free(temp.coeff);
}
```

Though we have discussed potential sources of bugs in connection with poly, they apply to other data structures as well whenever these data structures use some dynamic array to hold some information. Typical examples are those that represent matrices, vectors, polynomial factors and even multi-precision integers that we will define later. One should take enough care to deal with such data structures as recommended by the above examples on poly.

5 Matrices over finite fields

GFL provides the common routines for working with matrices over finite fields. The header file for these matrix manipulation routines is named matrix.h. As in the case of polynomials, GFL uses a dynamic array for representing the elements of a matrix. It defines the following two data structures.

```
typedef struct {              /* Data structure matrix */
    int row;                  /* Number of rows */
    int col;                  /* Number of columns */
    GFelement **element;  /* Pointer to 2-dimensional array of matrix elements */
} matrix;

typedef struct {              /* Data structure vector */
    int size;                 /* Vector size */
    GFelement *element;   /* Pointer to the array of vector elements */
} vector;
```

As with GFelement and poly, the definitions of matrix and vector avoid committing to a particular field. When one writes routines involving matrices and vectors, one must supply the GF_d over which the field operations should take place.

A vector in this paradigm is neither a row vector nor a column vector. It is just an array of GFelement. It is up to the user how she would like to view it. In some cases we use a vector as an ordered list (tuple) or even as an unordered set. There are certain routines, however, where GFL assumes specific structure on their vector arguments. Most notably, the linear equation solving routines like linSysSolve assume the vectors to be column vectors, that is, they solve systems of the form $A\mathbf{x} = \mathbf{b}$ (and not of the form $\mathbf{x}A = \mathbf{b}$ where vectors are treated as row vectors). Conversely, the routine vectorMatrixProd assumes that the vector concerned is a row vector. As one might guess, there is a matrixVectorProd too for dealing with column vectors. At any rate, we encourage GFL users to treat a vector as a *column vector*.

5.1 Reading and writing matrices and vectors

Similar to the polynomials, matrices and vectors can be created, assigned, copied, printed and also destroyed by a set of built-in library routines. Let's say that A and B are data of type matrix, u and v are vector. The routines newMatrix and newVector take no arguments and return a null matrix and a null vector respectively. One should use these routines for initializing matrices and vectors.[2] After initialization, the matrices and vectors are ready for getting assigned. readMatrix reads a matrix interactively from the keyboard. readVector does the same for vectors.

```
#include <gfl/field.h>
#include <gfl/matrix.h>

matrix A;
vector u;

/* Initialize matrices and vectors */
A = newMatrix(); u = newVector();

/* Read A and u interactively from stdin */
readMatrix(&A); readVector(&u);
```

The second way of assigning a matrix or a vector is: read from a suitable array of elements. The following routines with the stated prototypes can be used for this purpose.

```
void readMatrixFromArray ( matrix * , int , int , GFelement ** ) ;
void readVectorFromArray ( vector * , int , GFelement * ) ;
```

One should not confuse GFelement ** src with GFelement *src[] or with GFelement src[][]. They are different and cannot be interchanged. The first is a pointer to a pointer to GFelement, the second a pointer to arrays of GFelement, whereas the last is a two-dimensional array of GFelement. We refer to a good book on C (for example [14]) to settle all these C details. Here we demonstrate how one can do things when one uses GFL.

```
Matrix A;

GFelement **src = NULL;

src = (GFelement **)malloc(2, sizeof(GFelement *));
```

[2] Or at the minimum one should make the element pointer NULL. C may not do this during declaration. Later when one goes to free the memory, one might be in trouble with a non-initialized pointer. Why take risks after all?

```
src[0] = (GFelement *)malloc(3, sizeof(GFelement));
src[1] = (GFelement *)malloc(3, sizeof(GFelement));
src[0][0] = src[0][1] = src[0][2] = 0;
src[1][0] = 1; src[1][1] = 2; src[1][2] = 3;

A = newMatrix();

readMatrixFromArray(&A, 2, 3, src);
```

This assigns the matrix $\begin{pmatrix} 0 & 0 & 0 \\ 1 & 2 & 3 \end{pmatrix}$ to the matrix A. The procedure for vectors is similar. One may set a matrix or a vector to some specific types of matrices and vectors by using specific library calls. setZeroMatrix(&A, r, c) sets A to the *zero* matrix with r rows and c columns. setIdentityMatrix(&A, n) sets A to the $n * n$ *identity* matrix. For vectors, an all-zeros vector of size n is created and assigned to u by the call setZeroVector(&u, n). copyMatrix(&A, B) makes a verbatim copy of B to A. copyVector(&u, v) does the same with vectors. As discussed in connection with polynomials, these copy routines are different from assignments of the form A = B or u = v. Such assignments of structures involving dynamic arrays and/or pointers should always be avoided.

GFL provides routines for "destroy"ing matrices and vectors. When we are done with a matrix A and need it no longer, we may call destroyMatrix(&A) to free the memory associated with the element array of A. The counterpart for the vector u is destroyVector(&u).

To print a matrix A the routine

```
writeMatrix(A, K, fp, flag);
```

can be used, where K is a field descriptor (GF_d), fp is a file pointer (FILE *) opened in the 'write' or 'append' mode. If one supplies NULL as this argument, the output will be directed to stdout. The last element is a flag instructing printGFElement the way in which one wants to format the field elements. We have already described this in Section 3.4. Vectors are printed (as row vectors) by the analogous routine

```
writeVector(u, K, fp, flag);
```

5.2 Conversion between matrices and vectors

A vector can be viewed as a matrix with one of its dimensions equal to 1. GFL differentiates them by representing them as different data structures. It is, therefore, necessary to have, at our disposal, a set of routines that interchange matrices with vectors whenever the operations are permitted.

mtocv(&u, A)	Set u to a matrix A that has only one column
mtorv(&u, A)	Set u to a matrix A that has only one row
mctov(&u, A, i)	Set u to the ith column of A
mrtov(&u, A, i)	Set u to the ith row of A
cvtom(&A, u)	Treat u as a column vector and assign it to A
rvtom(&A, u)	Treat u as a row vector and assign it to A
vtomc(&A, u, i)	Treat u as a column vector and assign it to the ith column of A
vtomr(&A, u, i)	Treat u as a column vector and assign it to the ith row of A

5.3 Operations on matrices and vectors

The following table provides listing of the matrix operations provided by GFL and the syntax for using them. For efficient memory management, the assignments are effected by passing to the routines as the first argument a pointer to the matrix or vector being assigned. That is, instead of writing

```
A = matrixSum(B, C, K);
```

one should write

```
matrixSum(&A, B, C, K);
```

In the next table, we assume all vectors are *column vectors*. As before, we will denote matrices by uppercase letters (A, B, C etc.), vectors by lower case letters (u, v, w etc.), a finite field descriptor by K and scalars (i.e. field elements) by a, b etc.

Library call	**Effect**
equalMatrix(A,B)	checks if $A = B$
equalVector(u,v)	checks if $\mathbf{u} = \mathbf{v}$
zeroMatrix(A)	checks if A is the zero matrix
zeroVector(u)	checks if $\mathbf{u}$ is the zero vector
identityMatrix(A)	checks if A is the identity matrix
symmetricMatrix(A)	checks if A is a symmetric matrix
matrixTranspose(&A,B)	A is assigned the transpose of B
matrixSum(&A,B,C,K)	A is assigned $B + C$
vectorSum(&u,v,w,K)	u is assigned $\mathbf{v} + \mathbf{w}$
matrixDiff(&A,B,C,K)	A is assigned $B - C$
vectorDiff(&u,v,w,K)	u is assigned $\mathbf{v} - \mathbf{w}$
matrixProd(&A,B,C,K)	A is assigned BC
matrixVectorProd(&u,A,v,K)	u is assigned $A\mathbf{v}$
vectorMatrixProd(&u,v,A,K)	u is assigned $\mathbf{v}^t A$
scalarMatrixProd(&A,a,B,K)	A is assigned aB
scalarVectorProd(&u,a,v,K)	u is assigned $a\mathbf{v}$
matrixExp(&A,B,n,K)	A is assigned B^n for an integer n
matrixInv(&A,B,K)	A is assigned B^{-1}
matrixDet(A,K)	returns the determinant of A
singular(A,K)	checks if A is a singular matrix
matrixRank(A,K)	returns the rank of A
linIndep(va,n,K)	checks if the n vectors pointed to by the array va of vectors are linearly independent over K

5.4 Solving systems of linear equations

The routine linSysSolve solves a full-rank *square* system of linear equations and returns the solution vector to its first argument. It has the prototype

```
void linSysSolve ( vector * , matrix , vector , GF_d ) ;
```

Here it is assumed that the vectors are column vectors. That is, to solve a system $A\mathbf{v} = \mathbf{u}$ for the unknown vector **v**, this routine is called as

```
linSysSolve(&v,A,u,K);
```

The routine computes the LUP-decomposition of a matrix and then uses this decomposition to get the solution vector after forward and backward substitution. If one is interested in the LUP-decomposition of a matrix only, one may call the library routine LUPD. It computes L, U, and P for an invertible matrix A satisfying $PA = LU$ where P is a permutation matrix, L a lower-triangular matrix with 1's at the diagonal, and U an upper-triangular matrix. The syntax for the call is

```
LUPD(&B,pa,A,K);
```

After the call B holds the information for both L and U. The entries in B that are above the main diagonal (including the diagonal) are elements of U whereas those below the diagonal are elements of L (Note that L has 1's along the main diagonal). pa is an integer pointer (int *) that points to an array of integers that has the following relationship with P: The (i, j)-th element in P is 1 if and only if $k = i$ and $\mathsf{pa}[k] = j$ for some k.

Both linSysSolve and LUPD assume that the coefficient matrix is a *non-singular square matrix*. Indeed, LUPD returns a value that is true if and only if it has succeeded in computing the LUP-decomposition of the input matrix.

GFL provides separate routines for solving non-square systems of equations. First let's consider an overspecified system of m equations in n unknowns ($m \geq n$). Let us also assume that the rank of the coefficient matrix is n. In this case, the unique solution, if it exists, can be computed by calling the routine

```
overspLinSysSolve(&v, A, u, K);
```

where $\mathbf{v}$ is assigned the unique solution vector for the system $A\mathbf{x} = \mathbf{u}$. If the system is not consistent, overspLinSysSolve returns with an error message.

Next let's assume the case of underdetermined systems, i.e. the case when $m \leq n$, where m and n are the numbers of equations and unknowns respectively. We assume that the coefficient matrix has full row rank, namely m. In this case, there are more than one solution to the system, if $m < n$. The routine

```
underspLinSysSolve(&B, &v, &w, A, u, K);
```

gives all the solutions of $A\mathbf{x} = \mathbf{u}$ in the following way. Here A is an $m*n$ matrix, $\mathbf{x} = (x_0, x_1, \ldots, x_{n-1})^t$ is the vector of unknowns and **u** is a known $m*1$ vector. underspLinSysSolve fills an $m*(n-m)$ matrix B, an $m*1$ vector **v** and an $n*1$ vector **w**. $\mathbf{w} = (w_0, w_1, \ldots, w_{n-1})$ holds a permutation of the indices $0, 1, \ldots, n-1$. Then

$$(x_{w_0}, x_{w_1}, \ldots, x_{w_{m-1}})^t = \mathbf{v} + B(x_{w_m}, x_{w_{m+1}}, \ldots, x_{w_{n-1}})^t$$

Plugging in the values of $x_{w_m}, x_{w_{m+1}}, \ldots, x_{w_{n-1}}$ on the right side gives a solution for x_{w_0}, x_{w_1}, ..., $x_{w_{m-1}}$. Therefore, one gets all of the q^{n-m} solutions of $A\mathbf{x} = \mathbf{u}$ by putting all possible values of the vector $(x_{w_m}, x_{w_{m+1}}, \ldots, x_{w_{n-1}})$ where q is the cardinality of the field (K) over which the system is solved. If A is rank-deficient, underspLinSysSolve returns with an error message.

A special case (possibly the most important one) of underspLinSysSolve is sqLinSysSolve which solves a square system $A\mathbf{x} = \mathbf{u}$ where the $n*n$ matrix A is allowed to have rank $r < n$. The routine sqLinSysSolve(&B, &v, &w, A, u, K);

returns the rank r and produces the by-products: B – an $r * (n - r)$ matrix of GFelement, $\mathbf{v}$ – a vector of GFelement and of size r, and w – a vector of indices of length n. All the solutions can be obtained by plugging in values of the $n - r$ *independent* variables $x_{w_r}, x_{w_{r+1}}, \ldots, x_{w_{n-1}}$. Such a system might be inconsistent (when $r < n$). In that case, sqLinSysSolve returns with an error message. On the other hand, if one knows *a priori* that A has full rank ($r = n$), it is better to call the more efficient routine linSysSolve.

These generalized linear system solvers use one or both of the routines linIndepRows and linIndepCols to find out the indices of the linearly independent rows and columns of an $m*n$ matrix respectively. The former assumes $m \geq n$, whereas the latter $m \leq n$. Each returns the rank r of the matrix passed to it. Here are the prototypes of these function calls.

```
int linIndepRows ( vector * , matrix , GF_d ) ;
int linIndepCols ( vector * , matrix , GF_d ) ;
```

After returning from the routines the vector (whose reference is passed as the first argument) holds the desired indices. The first r entries of this vector are the indices of a set of linearly independent rows (resp. columns) of the input matrix. The rest are the indices of the remaining ones.

This leaves us with a few more general cases of linear systems. GFL does not provide built-in routines for solving such systems. Such systems do not occur frequently in practice. If the user faces such situations, she can use GFL built-ins to write her own procedures.

6 Irreducible polynomials

Irreducible polynomials are very important for the representation of finite field extensions. The header file irr.h declares routines for checking irreducibility of a polynomial over a finite field and for generating and listing irreducible polynomials. These routines are:

irreducible(f, K)

This routine checks whether the polynomial f is irreducible over the field K. The algorithm is based on Theorem 3.28 of [18].

findRandomIrrPoly(&f, K, d, flag)

This finds a random monic irreducible polynomial of degree d over the field K and stores the result in f. The flag specifies how to initialize the seed for the random number generator. The admissible values are

Value	**Type of seed**
1	Use the local time as seed
2	Use the value of the variable IRR_SEED_VAL as seed. IRR_SEED_VAL is defined in irr.h as an unsigned int and should be set to a desired value before findRandomIrrPoly is called.
anything else	Do not seed the random number generator

findRandomIrrPoly generates random polynomials of the given degree one by one and checks these polynomials for irreducibility till one irreducible polynomial is found. The function returns the number of iterations (that is, the number of polynomials checked) to find the irreducible polynomial.

findFirstIrrPoly(&f, K, d)

This function finds the lexicographically first (i.e. smallest) monic irreducible polynomial of degree d over the field K and stores the result in f. By lexicographic ordering of (the coefficients of) polynomials we mean the following. A polynomial $f(x) = x^d + a_{d-1}x^{d-1} + \cdots + a_1 x + a_0$ is said to precede $g(x) = x^d + b_{d-1}x^{d-1} + \cdots + b_1 x + b_0$, if the string $a_{d-1} \cdots a_1 a_0$ precedes the string $b_{d-1} \cdots b_1 b_0$ with respect to the ordering $0 \leq 1 \leq 2 \leq \cdots \leq \mathrm{card}(K) - 1$. findFirstIrrPoly returns the number of iterations necessary to find the first irreducible polynomial.

listAllIrrPoly(K, d, fp, flag)

This function prints all irreducible polynomials of degree d over the field K. fp is a file pointer (FILE *) for redirecting the output (NULL means stdout, as usual). The flag specifies the format in which the coefficients are to be printed. This has the same interpretation as the flag argument in printGFElement and writePoly. The function listAllIrrPoly returns the total number of irreducible polynomials found.

7 Some polynomial utilities

Now that we can represent matrices and vectors, we may describe the routines declared in the header file polyutil.h. The first two of these routines, however, do not need matrix or vector.

polyDerivative (&f, g, K)

sets f to the *formal derivative* of g. K is, as usual, the GF_d over which all field operations take place. The second routine we describe computes the minimal polynomial of an element in a finite field. Let $F \subseteq K$ be finite fields and $a \in K$. The call

minimalPoly(&f, a, K, F)

returns to f the minimal polynomial of $a \in K$ over F. Note that minimalPoly assumes that K is represented as an extension (not necessarily immediate) of F.

If A is a matrix with elements from a field K, the *characteristic polynomial* of A over K can be assigned to the poly f by the call

charPoly(&f, A, K)

Conversely, for a polynomial $f \in K[x]$, the *companion matrix* of f can be stored in the matrix A by the routine

compMatrix(&A, f, K)

Note that if $f(x) \in K[x]$ and if A is the companion matrix of f, then the characteristic polynomial of A is f. The *resultant* $\mathrm{Res}(f, g)$ of two polynomials $f, g \in K[x]$ is returned by the routine

polyRes(f, g, K)

This resultant is the same as the determinant of the *Sylvester matrix* of f and g. The *discriminant* of $f \in K[x]$ is returned by

polyDisc(f, K)

The discriminant of $f \in K[x]$ is given by $\mathrm{disc}(f) = (-1)^{d(d-1)/2}\mathrm{Res}(f, f')/a_d$, where $d = \deg(f)$, f' is the formal derivative of f and a_d is the leading coefficient of f.

8 Finding roots of univariate polynomials over finite fields

The header file <gfl/root.h> declares the GFL routines for finding the roots of univariate polynomials over a finite field. Each root-finding routine returns the number of roots found and stores these roots in the vector pointed to by the first argument. In this section, v is a vector, f is a poly, K and F are GF_d's and flag is a short int.

findRootES(&v, f, K)

This is the exhaustive-search technique that substitutes all elements of K for the unknown in the polynomial f to find out those elements $a \in K$ for which $f(a) = 0$. This works for all finite fields K, but is a reasonable strategy only when the cardinality of K is small (say, less than 100).

findRootBR(&v, f, K)

This is the randomized algorithm due to Berlekamp and Rabin [4]. This is quite efficient in practice and is recommended for fields of odd characteristic. The algorithm does not work for fields of characteristic 2.

findRootBT(&v, f, K, F)

This is the deterministic version of Berlekamp's trace algorithm. F can be any subfield of K (usually, the prime subfield of K). If $F = K$, this is equivalent to (but does more work than) findRootES. If F is of small cardinality, this algorithm runs quite efficiently. This works for all finite fields and is recommended for fields of characteristic 2.

findRootAM(&v, f, K, F)

This is the Berlekamp-Rumsey-Solomon algorithm [5] that uses affine multiples of polynomials. It computes the least q-affine multiple of f (where q is the cardinality of F), computes all the roots of this affine polynomial in K and finds which of these roots are those of f too. This works for finite fields of any characteristic and is efficient when q is small.

findRoot(&v, f, K, flag)

This calls one of the above algorithms depending on the value of flag. Let us summarize its behavior in the following table.

flag	**Behavior**
1	Calls findRootES
2	Calls findRootBR
3	Calls findRootBT with F = primeSubGF(K)
4	Calls findRootAM with F = primeSubGF(K)

anything else	if cardinality of K < SMALL_Q_BOUND
	    call findRootES
	else if characteristic of K is 2
	    call findRootBT with F = primeSubGF(K)
	else
	    call findRootBR

SMALL_Q_BOUND is #define-d in <gfl/root.h> as 100.

9 Factorizing univariate polynomials over finite fields

The header file <gfl/factor.h> defines the data structure polyFactor for holding the partial or complete factorization of a polynomial.

```
typedef struct {
    int nf;                /* Number of (distinct) factors */
    int *multiplicity;     /* Pointer to the array holding the multiplicities of the factors */
    poly *factor;          /* Pointer to the array of factors */
} polyFactor;
```

Similar to polynomials and matrices, polyFactor's need to be initialized by the routine newPolyFactor() before they can be used. And the routine destroyPolyFactor frees memory held by the arrays of a polyFactor. To print the factorization stored in a polyFactor, one may call the routine printFactors with four arguments. The first argument is the polyfactor itself, the second argument is the GF_d over which the polynomials are assumed to be defined, the third one is a file pointer (FILE *) opened for writing (the NULL value defaults to stdout), the last argument specifies the way of formatting the coefficients for printing (See Section 3.4).

```
#include <gfl/field.h>
#include <gfl/poly.h>
#include <gfl/factor.h>
polyFactor pf;
pf = newPolyFactor();
. . .
printFactors(pf, K, NULL, 6);
. . .
destroyPolyFactor(&pf);
```

The factorization routine is factorizePoly with the syntax

```
void factorizePoly ( polyFactor * , poly , GF_d ) ;
```

This is a randomized routine for univariate factorization over the specified GF_d. This calls the following three subroutines in succession.

```
void squareFreeFactorization ( polyFactor * , poly , GF_d ) ;
void distinctDegreeFactorization ( polyFactor * , poly , GF_d ) ;
void equalDegreeFactorization ( polyFactor * , poly , int , GF_d ) ;
```

These routines do as their names suggest, i.e., they respectively compute the square-free factorization, the distinct-degree factorization and the equal-degree factorization of a polynomial. Any arbitrary polynomial can be input to squareFreeFactorization, whereas for the last two, the polynomial should be square-free. In addition, the polynomial input to equalDegreeFactorization must be a product of irreducible polynomials of the same degree. The int parameter specifies this degree. The first two of these algorithms are deterministic, whereas the last one (which is based on [6, 7]) is probabilistic.

10 Affine polynomials

In this section we will assume that $F = \mathbb{F}_q \subseteq \mathbb{F}_{q^s} = K$ is a field extension (not necessarily immediate in terms of GFL representation). A q-affine polynomial $f \in K[x]$ is a polynomial of the form $f(x) = \alpha + \sum_{i=0}^{d} \alpha_i x^{q^i}$ where $\alpha, \alpha_i \in \mathbb{F}_{q^s}$. This affine polynomial can be represented using a variable f of data type poly in the following way.

f.degree = $d + 1$;
f.coeff[0] = α;
f.coeff[1] = α_0;
f.coeff[2] = α_1;
...
f.coeff[$d + 1$] = α_d;

Note that this is a dense representation of an affine polynomial.[3] Whenever we shall talk about affine polynomials (in this section or elsewhere), we will mean a representation in this form. The header file affine.h declares the following two routines. The first one

findLeastAffineMultiple (&f, g, K, F)

computes the least q-affine multiple of a general polynomial $g \in K[x]$ and stores the q-affine polynomial in the dense format in f. Here K and F are defined as before.

The second routine finds the roots of a q-affine polynomial $f \in K[x]$ in the following way. Let $a_0\gamma_0 + a_1\gamma_1 + \ldots + a_{s-1}\gamma_{s-1} \in K$ be a root of f, where $\gamma_0, \gamma_1, \ldots, \gamma_{s-1}$ is the composed basis of K over F. (See Section 3.4 for the definition of composed basis.) The following routine solves a system of s equations in the s unknowns $a_0, a_1, \ldots, a_{s-1}$ using sqLinSysSolve (See Section 5.4).

findRootAffinePoly (&A, &v, &w, f, K, F)

The routine computes A – an $r * (s - r)$ matrix of elements from F, $\mathbf{v}$ – a vector of elements from F and of size r and $\mathbf{w} = (w_0, w_1, \ldots, w_{s-1})^t$ – a vector of indices (between 0 and $s - 1$) of length s. The roots of f form an $(s - r)$-dimensional F-affine subspace. The unknowns $a_0, a_1, \ldots, a_{s-1}$ for points on this affine subspace are related as

$$(a_{w_0}, a_{w_1}, \ldots, a_{w_{r-1}})^t = A(a_{w_r}, a_{w_{r+1}}, \ldots, a_{w_{s-1}})^t + \mathbf{v}$$

Plugging in values of the independent variables $a_{w_r}, a_{w_{r+1}}, \ldots, a_{w_{s-1}}$ in the last equation gives us a solution for $a_0, a_1, \ldots, a_{s-1}$ and hence a root $a_0\gamma_0 + a_1\gamma_1 + \ldots + a_{s-1}\gamma_{s-1}$ of f in K. Since there are q^{s-r} distinct values of the independent variables, we get all of the q^{s-r} points in the $(s-r)$-dimensional F-affine subspace. In particular, when $r = s$, $\mathbf{v}$ gives the unique solution for $a_0, a_1, \ldots, a_{s-1}$.

11 Permutation polynomials

A polynomial $f(x) \in \mathbb{F}_q[x]$ is called a *permutation polynomial* of $\mathbb{F}_q$, if the function induced by f (i.e. the one that maps $a \in \mathbb{F}_q$ to $f(a) \in \mathbb{F}_q$) is bijective. GFL provides the routine (declared in permutation.h)

[3] Of course, the sparse representation of f in GFL is: f.degree = q^d, f.coeff[0] = α, f.coeff[q^i] = α_i for $i = 0, 1, \ldots, d$, and f.coeff[j] = 0; for $j = 1, \ldots, q^d$, $j \neq q^i$ for some i

int permPoly (poly f , GF_d K)

to check if the polynomial f is a permutation polynomial over K. This routine uses a randomized algorithm for checking permutation polynomials. It computes $\gcd(x^q - x, f - a)$ for random elements $a \in K = \mathbb{F}_q$. If for one such a the degree of the gcd is not equal to 1, f is not a permutation polynomial. On the other hand, if we repeat the process N times (i.e. with N random values of a), and if we get the degree of the above gcd to be 1 every time, we conclude that f is a permutation polynomial. A recent result by D. Wan [24] shows that when f is not a permutation polynomial and the degree of f is d, the probability that we get gcd of degree $\neq 1$ for a random value of a is at least $1/2d$. The number N is chosen such that the probability of failure (that is, $(1 - \frac{1}{2d})^N$) is sufficiently small.

12 Normal bases and N polynomials

The header file <gfl/normal.h> includes routines on testing and finding normal elements in finite fields.

int normal (GFelement , GF_d , GF_d)

Checks whether the given GFelement is a normal element of the first field over the second. The algorithm is based on Theorem 4.5 of [18].

int NPoly (poly , GF_d)

Checks whether the given polynomial is an N polynomial over the given field.

long listAllNormalElements (GF_d , GF_d , FILE * , short)

Lists all normal elements in the first field over the second. The file pointer is an already open file pointer (or NULL to imply stdout) for directing the output. The short int specifies how to format the field elements for printing. The routine returns the total number of normal elements found.

long listAllNPolys (GF_d , int , FILE * , short)

Lists all N polynomials of the given degree (the int parameter) over the given field. The last two arguments have the same meaning as in listAllNormalElements. The routine returns the total number of N polynomials found.

GFelement findNormalElementR (GF_d , GF_d , short)

Finds a normal element in the first field over the second by randomly choosing elements of the first field and checking if it is a normal element. This is a probabilistic polynomial time algorithm (See [10] for an analysis of the probability of failure). The last argument is a flag that specifies how to initialize (seed) the random number generator. It's value is interpreted as:

1	Use current time as seed
2	Use the value of NORMAL_SEED_VAL (an unsigned int declared in normal.h) as seed value
anything else	don't seed the random number generator

GFelement findNormalElementA (GF_d , GF_d , short)

Finds a normal element in the first field over the second using the algorithm that follows from Artin's lemma [2]. This is a better probabilistic algorithm than the previous one, when $q > 2s(s-1)$, where q is the cardinality of the smaller field and s is the degree of extension of the first field over the second. On the other hand, if $q \leq s(s-1)$, this algorithm may not generate any normal element at all. The last argument is again a directive for seeding the random number generator and has the same meaning as in the previous algorithm.

GFelement findNormalElementBDS (GF_d , GF_d)

Finds a normal element in the first field over the second using the algorithm due to Bach, Driscoll and Shallit [3]. This is a deterministic polynomial time algorithm. This uses two auxiliary routines which deserve specific mention on their own merit. The first one refines the factorization of $f^m g^n$ from a knowledge of the polynomials f, g and the positive integers m and n. The refined factorization is stored in the polyFactor pointed to by the first argument. The last argument is a GF_d over which all field operations take place.

polyPairRefine(&pf, f, g, m, n, K)

This routine is utilized in the general factor refinement routine for polynomials.

polyFactorRefine(&pf, ipf, K)

where ipf is the polyFactor holding the initial factorization and pf is a pointer to a polyFactor supplied for storing the refined factorization. K, as usual, specifies the ground field for these operations.

GFelement findNormalElement (GF_d , GF_d , short)

This routine calls one of the above routines for finding a normal element in the first field over the second depending on the value of the short flag supplied as the last argument. The flag is interpreted as follows:

flag	Action
1	Call findNormalElementR
2	Call findNormalElementA
3	Call findNormalElementBDS
anything else	if $q > 2s(s-1)$ call findNormalElementA else call findNormalElementBDS

where q and s have the same meanings as in the description of findNormalElementA.

13 Primitive elements

There are currently no known deterministic or randomized algorithms for finding a primitive element in a finite field or even for checking the primitivity of an element in a finite field, that run in time polynomially bounded by $\log q$, where q is the cardinality of the field. If we, however, know *a priori* the integer factorization of $q-1$, it is possible to design efficient algorithms for both these problems. The GFL header file primitive.h declares the following two routines: the routine

int primitive (GFelement , GF_d)

checks whether the first argument is primitive over the field with descriptor given by the second argument, whereas the routine

GFelement findPrimitiveElement (GF_d , short)

returns a random primitive element in a finite field (supplied as the first argument). Both these routines factorize $q-1$ before checking or finding primitive elements (where q is, again, the cardinality of the given field). The second argument of findPrimitiveElement is a directive to seed the random number generator and has essentially the same interpretation as we discussed in connection with the routine findNormalElementR. That is, if its value is 1, findPrimitiveElement seeds the random number generator by the current time. If the value is 2, the value of the unsigned int PRIMITIVE_SEED_VAL is used as the seed. For other values, the random number generator is not seeded at all.

Primitive elements can be used to speed up arithmetic in finite fields (See Chapter 6 of [18]). To see how let's assume that the fields $\mathbb{F}_p = F \subseteq K = \mathbb{F}_{p^s}$ are defined. Let g be a primitive element of K. For arbitrary elements $a, b \in K$, $a, b \neq 0$, let u and v be the discrete logarithms of a and b respectively with respect to g, that is $a = g^u$ and $b = g^v$. Then u and v are integers unique modulo $p^s - 1$. Then the product $a \cdot b \in K$ is g^w where $w = u + v (\bmod\, p^s - 1)$. Now let's assume that $v > u$. Then $a + b = g^u(1 + g^{v-u})$. If we know the discrete logarithm of $1 + g^{v-u} = g^{v'}$ (say), then we can calculate $a + b = g^{w'}$ where $w' = u + v' (\bmod\, p^s - 1)$.

In general, it is computationally very hard to find discrete logarithms in a finite field. Though GFL provides routines for computing discrete logarithms in finite fields (as we will see in a latter section), use of these routines does not speed up finite field arithmetic. Instead GFL provides facilities for creating and using tables of powers of a primitive element and discrete logs with respect to the same primitive element. These tables are stored as dynamic arrays in GF_info records (See Section 3.3). They are called primpower and disclog. GFL does not create them automatically. Instead a call of createPrimeGF or createExtGF makes them NULL. They can be created later by an explicit call like

```
createPrimTable(K, g)
```

Both the primpower and disclog tables are used for computing products in K. If one wants to use primitive elements to accelerate sums (and differences) too, one needs another table called *Zech's logarithm table* [13], that stores for u the discrete log of $1 + g^u$. This table is another dynamic array in GF_info struct with the name zechlt and can be explicitly created by the call

```
createZechTable(K)
```

after the primpower and disclog tables are created.

Each of these three tables must reside in the main memory and therefore their sizes are limited by the amount of RAM provided by the system. With current-day technology it is possible even with small personal computers to store tables for fields as large as 2^{16}. For larger fields we recommend the following strategy. Suppose we want to work over $\mathbb{F}_{3^{100}}$.[4] We create the tower of finite fields $F \subseteq K \subseteq L$, where $F = \mathbb{F}_3$, $K = \mathbb{F}_{3^{10}}$ and $L = \mathbb{F}_{3^{100}}$. We then create primitive power, discrete log and Zech logarithm tables for the intermediate field $K = \mathbb{F}_{3^{10}}$. This speeds up product in L by a factor of 100 and sum in L by a factor of 10. [5] This shows that maintaining these tables proves to be useful for large fields too.

Note that application of Zech's logarithm table does not make calculation of sums fast in fields of characteristic 2. This is because the routine for summation can handle many bits per logical instruction and table lookup cannot do better than that. This is why GFL does not allow the user to create Zech's logarithm table for these fields. The primpower and disclog tables can, however, be created as described above.

One might wonder what one has to do to notify GFL routines that the logarithm tables are ready for faster computation. The answer is that the user does not have to do anything to this effect. Every call of

[4] With the single-precision library, one cannot work with such "large" fields. That's why the multi-precision library is designed. This example, therefore, applies to this latter library. It, anyway, illustrates the concepts that we are trying to explain.

[5] These figures are theoretical. In practice, the speedup is less, but still sizable enough to justify our claim.

GFprod etc. checks whether the tables are ready or not by checking NULLity of the relevant pointers. If the tables are ready, it calls a set of routines that use table lookup.

Once the primitive tables are set up, they can be saved in hard disk files by the library calls

```
void savePrimTable ( GF_d , char * )
void saveZechTable ( GF_d , char * )
```

The first routine saves the primpower and disclog tables, whereas the latter saves Zech's table (zechlt). In either call, the second argument is a string implying the name of the file to save the information. Both these routines rewrites the files, if files of the given names already exist. In particular, when the user calls both these routines from her code, she must supply for storing the Zech logarithm table a filename different from that for storing the other two tables. The converse operation, i.e. reading the tables from hard disk files (instead of creating them afresh) can be done by

```
void readPrimTable ( GF_d , char * )
void readZechTable (GF_d , char * )
```

Here the second arguments are, again, the filenames where the tables were stored previously. Note that GFL does not store information regarding the representation of the field for which some tables are saved. As a result, one will get erroneous results, after one reads logarithm tables for another field (or the same field with different representation). GFL never complains that the user has made a mistake. It's up to the user to ensure that the tables she is reading are making enough sense.

Two more routines are waiting for us before we switch to the next topic. They are

```
destroyPrimTable ( GF_d )
destroyZechTable ( GF_d )
```

They, as their names suggest, free memory associated with the logarithm tables. One is expected to get a warning message if the logarithm tables are already NULL.

Now that we have described all the fields of the structure GF_info (that we introduced in Section 3.3), we can state how this info array is defined:

```
struct {                    /* Array holding info about created Galois fields */
    GFelement p;            /* characteristic */
    GFelement q;            /* cardinality */
    int extlev;             /* level of extension */
    GF_d extof;             /* descriptor of the field of which it is an algebraic extension */
    int extdeg;             /* extension degree over its immediate subfield */
    int totextdeg;          /* extension degree over prime subfield */
    GFelement *defpoly;     /* defining polynomial over its immediate subfield */
    long *primpower;        /* powers of a primitive element */
    long *disclog;          /* discrete log of elements w.r.to the primitive element */
    long *zechlt;           /* Zech's logarithm table to facilitate sum over finite fields */
} GF_info[MAX_FIELDS];
```

14 Basis utilities

We have seen that GFL uses a polynomial basis for representing an extension of a finite field. Once we call createExtGF, we fix a polynomial basis representation for the new extension field created. We have also seen that these polynomial bases for a tower of fields can be combined to get what we called a composed basis, for representing field extensions that are not immediate. In general, however, there are other bases for finite field extensions. As an example, if $\beta \in K \supseteq F$ is a normal element of K over F, then $\beta, \beta^q, \beta^{q^2}, \ldots, \beta^{q^{s-1}}$ is a basis of K over F, where q is the cardinality of F and $[K : F] = s$. GFL provides facilities for conversion of an element written in one basis to one written in another. The header file for these routines is basis.h.

The first routine we state is

int basis (vector , GF_d , GF_d)

It checks if the elements of the vector form a basis of the first field (the second argument) over the second (the third argument). Thus if $F \subseteq K$ is a field extension defined (not necessarily immediate) and if the vector in the first argument comprises of elements $\alpha_0, \alpha_1, \ldots, \alpha_{s-1} \in K$, the routine checks if these s elements form an F-basis of K.

Now suppose we have defined field extensions $F \subseteq K \subseteq L$ by polynomials $K = F/{<}f{>}$ and $L = K/{<}g{>}$ with $\deg f = s$ and $\deg g = t$. Let $\alpha \in K$ be a root of $f(x)$ and $\beta \in L$ a root of $g(x)$. Then GFL uses the polynomial basis $1, \alpha, \alpha^2, \ldots, \alpha^{s-1}$ of K over F and the polynomial basis $1, \beta, \beta^2, \ldots, \beta^{t-1}$ of L over K. Now the set of st elements

$$1, \alpha, \ldots, \alpha^{s-1}, \beta, \beta\alpha, \ldots, \beta\alpha^{s-1}, \ldots, \beta^{t-1}, \beta^{t-1}\alpha, \ldots, \beta^{t-1}\alpha^{s-1}$$

forms an F-basis of L. We have defined this basis to be the *composed basis* of L over F. GFL uses this basis when it has to view elements of L as vectors over F. In general, this composed basis is not a polynomial basis of L over F. There are, however, situations where we want a polynomial basis of L over F. Let's see how we can find that. Note that we have described a routine that computes a monic irreducible polynomial of a given degree over a given finite field. Let us, say, that $h(x)$ is a polynomial in $F[x]$, that is monic, irreducible over F and has degree st. We know from the theory of finite fields that h splits completely over L. Let γ be a root of h in L. Then $1, \gamma, \gamma^2, \ldots, \gamma^{st-1}$ is a polynomial basis of L over F. It's clear that such construction of polynomial basis works for an arbitrary extension of finite fields. (In particular, if the extension is immediate, the polynomial basis is available from the representation of that extension itself.) GFL provides the following routine that does all these for getting a polynomial basis of a field over a subfield of it.

void polyBasis (vector * , poly * , GF_d , GF_d , short)

This routine computes a polynomial basis of the first field (the third argument) over the second (the fourth argument). The polynomial that defines the polynomial basis (h in the above example) is returned through the second argument, whereas the basis elements (powers of γ in the example) are returned in the element array of the vector (the first argument). The short flag is simply transfered to findRandomIrrPoly as a directive to seed the random number generator.

Now that we have different types of bases of a finite field over a subfield (polynomial, composed, normal, or even an arbitrary one as certified by the routine basis), there should be a provision for conversion between the bases. This is done by *transformation matrices*. Suppose $F = \mathbb{F}_q \subseteq \mathbb{F}_{q^s} = K$ are finite fields. Suppose $\underline{\alpha} = \{\alpha_0, \alpha_1 \ldots, \alpha_{s-1}\}$ and $\underline{\beta} = \{\beta_0, \beta_1, \ldots, \beta_{s-1}\}$ are two F-bases of K. Let an element $a \in K$ be represented in these two bases as

$$a = a_0\alpha_0 + a_1\alpha_1 + \ldots + a_{s-1}\alpha_{s-1} = b_0\beta_0 + b_1\beta_1 + \ldots + b_{s-1}\beta_{s-1}$$

where $a_i, b_j \in F$. The transformation matrix from the basis $\underline{\alpha}$ to the basis $\underline{\beta}$ is an invertible $s * s$ matrix T with entries from F such that

$$(b_0, b_1, \ldots, b_{s-1})^t = T(a_0, a_1, \ldots, a_{s-1})^t$$

GFL provides the following routines for computing the transformation matrices from one basis to another. The first five of them work only for *immediate* extensions. The polynomial bases that are used for representing these extensions play the central role for conversion between bases. Note that for immediate extensions these polynomial bases are same as composed bases. In each of the following calls the transformation matrix is stored in the matrix pointed to by the first argument. Bases are passed to these routines via vector's (with entries from the extension field). When normal bases are involved, it is sufficient to pass a normal element. (Other basis elements are computed by raising the normal element to suitable powers.) The GF_d supplied to these routines is that of the extension field, that is, a transformation matrix is computed for conversion between two bases of this field regarded as a vector space over its immediate subfield.

void gtopTransMatrix (matrix * , vector , GF_d)
: transformation matrix from a general basis to the polynomial basis

void ptogTransMatrix (matrix * , vector , GF_d)
: transformation matrix from the polynomial basis to a general basis

void gtogTransMatrix (matrix * , vector , vector , GF_d)
: transformation matrix from a general basis to another general basis

void ntopTransMatrix (matrix * , GFelement , GF_d)
: transformation matrix from a normal basis to the polynomial basis

void ptonTransMatrix (matrix * , GFelement , GF_d)
: transformation matrix from the polynomial basis to a normal basis

A similar set of routines for general extensions (i.e. extensions that are not necessarily immediate) is in order. As above, the result is stored in the matrix pointed to by the first argument. Bases are passed as vector's with elements from the extension field. Two GF_d's are passed to these routines: the first one is that for the extension field and the second one is that for the subfield.

void gtocTransMatrix (matrix * , vector , GF_d , GF_d)
: transformation matrix from a general basis to the composed basis

void ctogTransMatrix (matrix * , vector , GF_d , GF_d)
: transformation matrix from the composed basis to a general basis

void atoaTransMatrix (matrix * , vector , vector , GF_d , GF_d)
: transformation matrix from a general (arbitrary) basis to another general (arbitrary) basis

void ntocTransMatrix (matrix * , GFelement , GF_d , GF_d)
: transformation matrix from a normal basis to the composed basis

void ctonTransMatrix (matrix * , GFelement , GF_d , GF_d)
: transformation matrix from the composed basis to a normal basis

These basis conversion routines are used extensively in finding isomorphism between finite fields, in the computation of discrete logarithms in finite fields, and so on.

15 Isomorphism between finite fields

We know from the theory of finite fields that two finite fields are isomorphic if and only if the fields have the same cardinality. GFL allows one to define two fields of the same cardinality using two different sequences of calls to createExtGF. For example, one might create F to be a prime field of cardinality p, say. Then one finds an irreducible polynomial f of degree s over F and uses it to create the extension field $K = F/<f>$. A second level extension $L = K/<g>$ can be created by an irreducible (over K) polynomial $g \in K[x]$ of degree t. We then have $[L : F] = st$. Suppose we compute an irreducible polynomial h of degree st over F and use this to extend F in order to get $L' = F/<h>$. We have $L \cong L'$. But in this case an element in L has a representation different from that of its image in L' (under some isomorphism between L and L'). The user should, therefore, have a mechanism that converts elements from one representation to another. GFL provides the procedure findIsoMatrix declared in the header file iso.h to compute such transformations between isomorphic fields in the form of a matrix. The prototype for the call is

```
void findIsoMatrix ( matrix * , GF_d , GF_d )
```

The GF_d's supplied are those of two fields of the same cardinality. Let us call these fields L and L' respectively. Let $\mathbb{F}_p$ be their prime subfield and $[L : \mathbb{F}_p] = [L' : \mathbb{F}_p] = m$. The routine returns through the first argument an $m * m$ matrix T that can be used as follows. Let the representation of an element $a \in L$ in the composed basis of L over $\mathbb{F}_p$ be the m-tuple $(a_0, a_1, \ldots, a_{m-1})$. If we represent m-tuples by vector's of size m, then the representation of a (more correctly, of the isomorphic image of a in L') in the composed basis of L' over $\mathbb{F}_p$ is to be computed by the matrix-vector product $T(a_0, a_1, \ldots, a_{m-1})^t$.

The routine findIsoMatrix works as follows. Let us use the notations introduced in the last paragraph. findIsoMatrix finds polynomial basis representations of L and L' over $\mathbb{F}_p$ defined by the *same* irreducible polynomial of degree m over $\mathbb{F}_p$. That is, it computes the ctogTransMatrix T_1 of L over $\mathbb{F}_p$ and the gtocTransMatrix T_2 of L' over $\mathbb{F}_p$. The *isomorphism matrix* from L to L' is then $T = T_2 T_1$.

findIsoMatrix is probabilistic. Though polynomial-time deterministic algorithms exist to solve this problem (See, for example, [15]), the probabilistic variant is, in practice, more efficient than the deterministic ones.

16 Discrete logarithms

Computing discrete logarithms in a finite field is, till date, a hopelessly complex problem. No polynomial time algorithms are known to solve the problem, even when one allows randomization. The best known algorithm is the *index calculus method* that has *sub-exponential* running time. (See [25] for a generic description of the index calculus method, [1, 17] for application of this method to compute discrete logarithms in prime fields $\mathbb{F}_p$, and [21] for the same in arbitrary finite fields $\mathbb{F}_{p^s}$.) GFL implements this method for finding discrete logarithms in fields of both prime and prime power cardinalities. One should include the header file dlog.h for the data structures and procedure declarations that are necessary for computing discrete logarithms using GFL.

GFL assumes that we are calculating the discrete log of an element with respect to some *primitive element*. In practice, it is usual to compute discrete logs with respect to some element of "large" order in the multiplicative group of the finite field. To avoid unnecessary complications, we take primitive elements as the base of discrete logarithms. Throughout this section we will denote by g the primitive element with respect to which we want to take discrete logarithms.

The index calculus method proceeds in two stages. In the first stage, we decide a suitable *factor base* and compute logarithms of elements in the factor base. In the second stage we calculate individual logarithms by finding elements that "factor completely" over the factor base. The first stage is a sort of "precomputation" and need be carried out only once for calculating any number of individual logs in the second stage. As a reason, GFL provides separate calls for these two stages. In addition, the logarithms of elements in the factor base, once computed, can be saved in hard disk files so that they can be reused later. This saves time for recomputation of the first stage.

16.1 Data structures

Before we go into the details of the library calls, we need to introduce some data structures for holding information about the factor base. GFL uses different data structures for fields of different extension levels.

```
/* Factor base for prime fields, i.e. fields of extension level = 0 */
typedef struct {
    GFelement p;        /* characteristic of the field in which discrete log is taken */
    GFelement gen;      /* base to which discrete log is taken */
    int fbsize;         /* number of primes in the factor base */
    GFelement *base;    /* elements of factor base */
    long *baselog;      /* discrete logs of factor base elements */
} factorBase1;

/* Factor base for non-prime fields of extension level = 1 */
typedef struct {
    GFelement p;        /* characteristic of the field in which discrete log is taken */
    GFelement gen;      /* base to which discrete log is taken */
    int maxdeg;         /* maximum degree of irreducible polynomials in the factor base */
    int fbsize;         /* number of elements in the factor base */
    int nprime;         /* number of primes between 2 and p-1 */
    GFelement *base;    /* elements of factor base */
    long *baselog;      /* discrete logs of factor base elements */
} factorBase2;

/* Factor base for non-prime fields of extension level > 1 */
typedef struct {
    GFelement p;        /* characteristic of the field in which discrete log is taken */
    matrix ctop;        /* matrix for composed-to-polynomial basis transformation */
    GFelement gen;      /* base to which discrete log is taken */
    int maxdeg;         /* maximum degree of irreducible polynomials in the factor base */
    int fbsize;         /* number of elements in the factor base */
    int nprime;         /* number of primes between 2 and p-1 */
    GFelement *base;    /* elements of factor base */
    long *baselog;      /* discrete logs of factor base elements */
} factorBase3;
```

In order to see what the fields of the above structures mean, let us first consider the case of a prime field $\mathbb{F}_p$. This is a field that can be represented as a field of extension level 0. The structure factorBase1 is applicable in this case. The meanings of the struct components p and gen should be clear. We explain the other components of factorBase1. For prime fields, the factor base consists of the prime numbers less than a given bound. These primes are stored in increasing order of magnitude in the dynamic array pointed to by base. The discrete logs of these elements are stored in the respective order in the dynamic array pointed to by baselog. The size of these arrays is stored in the int component fbsize. This completes the interpretation of the structure factorBase1.

Now let us look at a finite field of cardinality $q = p^s$, where p is a prime and $s > 1$. Let us further suppose that such a field is represented as an immediate extension of the prime field $\mathbb{F}_p$, that is, $\mathbb{F}_q = \mathbb{F}_p/{<}f{>}$, where f is an irreducible polynomial over $\mathbb{F}_p$. Elements of $\mathbb{F}_q$ are represented as polynomials of degree less than s over $\mathbb{F}_p$. Arithmetic is performed on these elements using the polynomial arithmetic of $\mathbb{F}_p$ modulo the defining polynomial f. In this case, the factor base comprises of primes between 2 and $p - 1$ and monic irreducible polynomials of degree less than or equal to a predefined bound (the component maxdeg in factorBase2). nprime stores the number of primes between 2 and $p-1$, fbsize is the total number of elements in the factor base (including nprime primes), base points to the dynamic array holding (in increasing order of magnitude) those GFelement's that correspond to the primes and irreducible polynomials in the factor base, and finally baselog is a pointer to the array of logarithms of these factor base elements.

Finally let's consider a field of extension level greater than 1, i.e. a field that is neither prime nor an immediate extension of a prime field. The relevant data structure is factorBase3 for such a field. In this case the default representation of the field over its prime subfield is the composed basis representation. We force a polynomial basis representation of the field over its prime subfield and store in the matrix component ctop of factorBase3 the corresponding transformation matrix for converting elements from the composed basis representation to the polynomial basis representation. All other components of factorBase3 have the same interpretations as those of factorBase2.

In what follows we will call the numbers 1, 2 or 3 in factorBase1 etc. the *type* of the factor base. It will often be useful to talk about a data type factorBase. GFL does not define a data type with this name. We use this name as a symbol that can be substituted by factorBase1 or factorBase2 or factorBase3 depending on the situation.

When working in a field of some extension level (say, l), one might use a data structure that is meant for fields of extension level greater than l. This practice is, however, discouraged in view of efficiency considerations and storage requirements. Thus, for example, suppose that while dealing with a field of extension level 1, we decide to use factorBase3. We, then, need memory for storing the ctop matrix which is the identity matrix in this case. In addition, the composed-to-polynomial basis conversions increase the running time by computing trivial matrix-vector products. Both these could be avoided, if we used factorBase2 instead of factorBase3.

16.2 Stage 1: Creating factor base

We have told that we use the "undefined" data type factorBase as a generic name for the "defined" data types factorBase1, factorBase2 and factorBase3. The *type* of factorBase will be evident from the programming context. In a similar manner, we will use "generic" names for subroutines that work with factor bases. Subroutines with these generic names are *not* defined. One has to append 1, 2 or 3 to these names to get the names of the defined functions that are applicable to factorBase of type 1, 2 and 3 respectively. For example, a factor base of type $<\tau>$ can be created by the call createFactorBase$<\tau>$, for $<\tau>$ =1, 2 or 3. Instead of writing the symbol $<\tau>$ or one of its values whenever we refer to this routine, we will use the generic name createFactorBase. It should be understood that createFactorBase1 creates a factorBase of type 1, createFactorBase2 creates one of type 2 and so on. With this convention, we state a generic description of the routines for manipulating factorBase's.

Generic name	Behavior
newFactorBase	Create a null factorBase
destroyFactorBase	Free memory associated with a factorBase
createFactorBase	Create a factorBase (this includes deciding elements for a factorBase and computing logarithms of these elements)
saveFactorBase	Save factorBase in hard disk for future reuse
readFactorBase	Read factorBase from hard disk

The prototypes of these calls are

```
factorBase newFactorBase ()
void destroyFactorBase ( factorBase * )
void saveFactorBase ( factorBase , char * )
void readFactorBase ( factorBase * , char * )
void createFactorBase1 ( factorBase1 * , GF_d , GFelement , long )
void createFactorBase2 ( factorBase2 * , GF_d , GFelement , int )
void createFactorBase3 ( factorBase3 * , GF_d , GFelement , int )
```

The second parameter (char *) of saveFactorBase or readFactorBase is the name of the hard disk file where we want to save or from where we want to read data of a factorBase. The prototypes for createFactorBase are different for different types of factor bases. In each case, the first argument is a pointer to the appropriate factorBase where the result will be stored, the second argument is the descriptor of the field for which the factor base is created and the third element is a primitive element of this field with respect to which discrete logs are taken. The last argument of createFactorBase1 is an upper bound on the primes in the factor base, i.e. the factor base will comprise of primes less than or equal to this bound. For createFactorBase2 and createFactorBase3, the last argument is the largest degree of irreducible polynomials in the factor base. This degree is the same as the component maxdeg in factorBase2 or factorBase3.

16.3 Stage 2: Computing discrete logs

Now that we know how to create a factorBase, we can use it for computing discrete logarithms. The routines are named dlog1, dlog2 and dlog3 depending on the type of the factorBase we are passing to them. The generic description of dlog is

```
long dlog ( GFelement , GF_d , factorBase )
```

The first argument is the element of which the discrete log is to be computed, the second argument is the GF_d of the field in which field arithmetic will be carried out and the third argument is the factorBase already created or read from hard disk. dlog does not check whether the factorBase is meant for the field passed through the second argument. If one creates a factorBase for some field and wants to use this factorBase for computing discrete logs in another field, dlog won't complain, but one should be prepared to get meaningless (or no) results in such a situation.

16.4 An example

Let's illustrate the concepts introduced in the last few sections by an example. We want to compute discrete logs in the field $\mathbb{F}_{3^7}$. We define this field as an immediate extension of the prime subfield $\mathbb{F}_3$.

(Since 7 is a prime, this is the only possible representation of $\mathbb{F}_{3^7}$.) We must use type 2 factor bases and the corresponding routines.

```
#include <stdio.h>
#include <gfl/field.h>
#include <gfl/poly.h>
#include <gfl/irr.h>
#include <gfl/primitive.h>
#include <gfl/dlog.h>

GF_d F, K;
GFelement p = 3, g, a;
factorBase2 fb;
poly f;
int maxdeg = 3;

/* Initialize data structures */
f = newPoly(); fb = newFactorBase2();

/* Create the prime field */
F = createPrimeGF(p);

/* Get random irreducible polynomial of degree 7 over F */
findRandomIrrPoly(&f, F, 7, 1);

/* Use this polynomial to define the extension field */
K = createExtGF(F,f);

/* Find random primitive element in K */
g = findPrimitiveElement(K, 1);

/* Create factor base of type 2 for calculating discrete logs in K with respect to the primitive
element g */
createFactorBase2(&fb, K, g, maxdeg);

/* Compute individual discrete logs */
a = dlog2(1, K, fb);
a = dlog2(10, K, fb);
a = dlog2(100, K, fb);
a = dlog2(1000, K, fb);
a = dlog2(1998, K, fb);

/* The following call gives an error message, because log of zero is undefined */
```

```
a = dlog2(0, K, fb);

/* Free memory associated with poly's and factorBase's */
destroyPoly(&f); destroyFactorBase2(&fb);
```

17 Multi-precision Integers

So far we have discussed about GFL routines where the cardinality of a finite field is assumed to fit in a single long int. More specifically, if N is the largest positive integer representable by a (signed) long int, the characteristic p and the cardinality q of a finite field must satisfy the inequality

$$p^2 + p < N, \ \ q < N$$

for these routines to work for that field without possibility of overflow.[6]

In this section, we introduce routines that overcome this restriction. Here we will work with multi-precision integers that can span across several long int's. We will first describe how such multi-precision integers are represented in computer memory. Then we will talk about how one can do arithmetic operations on these multi-precision integers. In the next section, we will see what changes need to be incurred in the GFL routines because of this change in the representation of numbers. We will say that these modified GFL routines together with the multi-precision integer manipulation routines constitute the "Multiprecision Galois Field Library" or "MPGFL" in short.

Before we delve ourselves into the gory details of MPGFL, it is useful to point out that MPGFL routines have higher overhead than their single-precision counterparts. It is, therefore, practical to use the single-precision GFL instead of MPGFL, when one is sure that the cardinalities of the finite fields over which one intends to work, satisfy the above inequalities. For example, for applications that deal with polynomials of large degrees over small finite fields, the single-precision GFL turns out to be the better choice. Finally notice that it is *not* possible to use the two libraries simultaneously. Each library implements the same set of routines keeping in mind its own representation of numbers.

17.1 Representation of multi-precision numbers

The basic data type mpint is defined in the header file <mpgfl/int.h> as

```
typedef struct {        /* Multi-precision integer */
    char sign;          /* '+' for positive numbers, '-' for negative numbers, ' ' for zero */
    int size;           /* Number of longs needed to represent the number */
    long *word;         /* link to the array of longs holding the number */
} mpint;
```

That is, an mpint defines a representation of *signed* integers of *arbitrary* length. The first field of the struct indicates the sign of the integer – '+', '-' or ' ' (space) according as whether the integer it holds is positive, negative or zero. The second field (size) is the exact number of long int necessary to hold the multi-precision integer and the third field (word) is a pointer to a dynamic array of long int holding the fragments of the (absolute value of the) integer.[7]

[6] In a machine with 32 bit long, it is perfectly safe to work with $p \le 2^{15}$ and $q \le 2^{30}$. Since $N = 2^{31} - 1$ in this case, slightly larger values of p and q might not create troubles. But we advise you to work in the safe zone.

[7] size should ideally be the same as the amount of memory (in terms of the number of long's) allocated to the pointer word. In practice, there is no harm (apart from having some unused allocated memory), if size is less than this amount of memory. On the other hand, one might be in trouble, if size indicates a larger value than the actual memory allocated to word.

Let's first consider an mpint n that holds a non-zero integer n. n represents n as a number in base RADIX. That is, each element in the array n.word is a number between 0 and $\mathsf{RADIX}-1$, both inclusive. n.word[0] is the least significant word and n.word[n.size-1] is the most significant word of n. One should have $\mathsf{n.word}[\mathsf{n.size}-1] \neq 0$. In MPGFL implementation, RADIX is a power of 2, specifically, $\mathsf{RADIX} = 2^{\mathsf{MAX\_BIT}}$. This means that each word of n.word holds MAX_BIT bits of n, n.word[0] the least significant MAX_BIT bits and n.word[n.size-1] the most significant MAX_BIT bits of n.[8] With the assumption that a long int has 32 bits, and that the data type double has 52 bit precision, <mpgfl/int.h> defines the entities RADIX and MAX_BIT as

```
#define MAX_BIT 25
#define RADIX 33554432
#define RADIX_MINUS_ONE 33554431
```

The third definition means what its name suggests. This quantity is used often in the MPGFL routines, so that we found it convenient to have a separate definition for $\mathsf{RADIX}-1$. If some machine supports larger long, say of length 64 bits, and larger double, these values should go fine. But, of course, one can improve the efficiency of the routines by increasing these values, so that an mpint can use more compact representation of integers. The only thing to keep in mind is that if N is the largest positive integer representable with full precision by a double, and if M is the largest positive integer representable by a signed long int, then RADIX must satisfy

$$\mathsf{RADIX} < M, \;\; \mathsf{RADIX}^2 + \mathsf{RADIX} < N$$

Thus, for example, if one has 64-bit longs and 128-bit double with, say, 100 bit precision, it's safe to define

```
#define MAX_BIT 49
#define RADIX 562949953421312
#define RADIX_MINUS_ONE 562949953421311
```

On the other hand, if some machine has long that is less than 32 bits long, one must change these three macros. For example, if long is 16 bits and double is 32 bits with 25 bit precision, one may define

```
#define MAX_BIT 12
#define RADIX 4096
#define RADIX_MINUS_ONE 4095
```

In the rest of this chapter, we will assume default values for MAX_BIT, RADIX and RADIX_MINUS_ONE (that is, $\mathsf{MAX\_BIT} = 25$ etc.). We will also use the symbol R to represent (the value of) RADIX.

There are three other macros defined along with the above three macros. These need also be changed if RADIX etc. are changed. These new macros are

```
#define HALF_MAX_BIT 13
#define HALF_RADIX_MINUS_ONE 8191
#define BIT_DIFF 1
```

[8] The number of bits in n need not be an integral multiple of MAX_BIT. In this case, n.word[n.size-1] might be viewed as being padded by leading 0's to make it a number of MAX_BIT bits.

where HALF_MAX_BIT is $\lceil$MAX_BIT$/2\rceil$, HALF_RADIX_MINUS_ONE is $2^{\text{HALF\_MAX\_BIT}} - 1$ and BIT_DIFF is $2 *$ HALF_MAX_BIT $-$ MAX_BIT.

Before ending this section, we mention the representation of 0 (zero) in this system. An mpint n, representing this special number must have

```
n.sign = ' ';
n.size = 1;
n.word[0] = 0;
```

Note that this is the only special case where we allow n.word[n.size - 1] = 0.

17.2 Basic operations with multi-precision integers

In order to use an mpint, one must first initialize it.

```
#include <mpgfl/int.h>
mpint n;
...
/* Initialize n */
n = newInt();
/* Now use n */
```

Or at the least one must make the word pointer NULL. This prevents possibilities of future memory hazards as explained in connection with poly, matrix etc.

An mpint may be assigned manually by allocating memory to the pointer word and then assigning suitable values to the elements of this array. The following example assigns the mpint n the integer 11223344556677889900998877665544332211. Note that

$$\begin{aligned} 11223344556677889900998877665544332211 &= 8853657R^4 + 25051344R^3 \\ &\quad + 6227312R^2 + 31737219R + 17261491 \end{aligned}$$

where $R = 2^{25} = 33554432$.

```
#include <mpgfl/int.h>

mpint n;

main() {

    n = newInt();

    n.sign = '+';       /* set the sign of n */
    n.size = 5;       /* set the size of the word array holding the value of n */

    /* allocate memory for n.word */
    n.word = (long *)malloc(5 * sizeof(long));
```

```
    /* assign individual values to the array elements */
    n.word[0] = 17261491;
    n.word[1] = 31737219;
    n.word[2] = 6227312;
    n.word[3] = 25051344;
    n.word[4] = 8853657;

    /* print n in two ways */

    /* print n as a decimal number */
    writeInt(n, stdout); printf("\n");

    /* print the decimal values of the word elements */
    showInt(n, stdout); printf("\n");
}
```

This program outputs

```
11223344556677889900998877665544332211
+[8853657][25051344][6227312][31737219][17261491]
```

There are several other methods by which an mpint can be assigned. The prototype of these routines are listed below.

```
void longToInt ( mpint * , long ) ;        /* Assign a long int to an mpint */
void assignInt ( mpint * , char * ) ;      /* Assign a string to an mpint */
void readInt ( mpint * , char * ) ;        /* Read an mpint from stdin */
void copyInt ( mpint * , mpint ) ;         /* Copy one mpint to another */
```

The first (longToInt) can be used for assigning small integer values (i.e. values that can fit a long int) to an mpint. assignInt assigns a character string to an mpint. This string must consist of only numeric characters (0. . . 9) optionally preceded by a sign character (+ or -). No spaces are allowed anywhere. If assignInt finds a non-numeric character where it expects a digit, it replaces the same by 0 after outputting a warning that it has done so. The routine readInt, on the other hand, reads a string from stdin and assigns that string to an mpint by calling assignInt. The second argument to readInt is a character pointer that points to a string which is printed by the routine as a prompt before the numeric string is input. Finally, copyInt copies an mpint (the second argument) to another mpint (the one pointed to by the first argument). Note that copyInt(&n, m) is not the same as n = m, because the former makes a *copy* of the word array of m to the word array of n, whereas in the latter, the pointer to the word array of m is copied to the pointer to the word array of n, so that subsequent changes in n.word[i] also changes m.word[i] for all integers i. copyInt outputs a warning message, when an attempt is made to copy an mpint to itself.

There are two routines that generate random mpint. These are

```
void randInt ( mpint * , int , short ) ;
void randRes ( mpint * , mpint ) ;
```

randInt returns in the first argument a random multi-precision integer whose bit-length is exactly equal to the second argument. The third argument specifies how to seed the random-number generator. If it is supplied a value of 1, the current time is used as the seed-value. If this argument is 2, randInt looks up the value of the unsigned int INT_SEED_VAL as the seed for the random number generator. For all other values of the third argument, the random number generator is not seeded at all. The second routine, say as in the call randRes(&n, m), assigns n a random number between 0 and $|m| - 1$, both inclusive.

We have already described two output routines for printing an mpint. These are

```
int writeInt ( mpint , FILE * ) ;
int showInt ( mpint , FILE * ) ;
```

Each of these two routines takes two arguments – the first is the mpint to be printed; the second is a FILE pointer already opened for writing (stdout, stderr or the user's own FILE pointer for writing to a disk file). If one passes NULL as this argument, the output is sent to stdout. The former routine (writeInt) prints an mpint in the decimal notation. The other routine (showInt) prints the sign of the mpint followed by the individual words of the mpint. The most significant word is printed first and the least significant last. Both these routines return the number of characters printed. Neither prints a trailing newline character.

MPGFL supports a routine that converts an mpint to a long int.

```
long intToLong ( mpint ) ;
```

This conversion makes sense if and only if the value of the mpint can fit in a single long int. If the situation is otherwise, intToLong returns a junk value. It does not check whether the conversion resulted in an overflow.

The word array of an mpint (if allocated memory) holds some space. If we have shortage of memory and we no longer need the value of an mpint, we may free the memory associated with the word array of the mpint. The built-in routine destroyInt can be used for this purpose.

```
void destroyInt ( mpint * ) ;
```

After the call destroyInt(&n), n.word becomes NULL and n.size becomes 0. This means that n now represents no valid integer (not even 0).

17.3 Comparing multi-precision integers

The routine

```
compint(n, m);
```

compares two mpint's n and m and outputs an int depending on the comparison result as follows:

Condition	**Return value**
$n = m$	0
$n > m$	1
$n < m$	-1

Some special comparisons can be made by calling one of the following routines.

Routine	returns *true* iff
zeroInt(n)	$n = 0$
positiveInt(n)	$n > 0$
negativeInt(n)	$n < 0$
unityInt(n)	$n = 1$
negUnityInt(n)	$n = -1$

17.4 Arithmetic operations on multi-precision integers

MPGFL provides a complete set of routines for doing arithmetic with mpint. We summarize the functionality of these operations in a small table. It is assumed that n, m, t, r and s are data of type mpint and the values they hold are respectively n, m, t, r and s. Similarly a is a variable of type long int and its value is a. As in the case of polynomials and matrices, the reference to an mpint need be passed to these routines, when we want to modify the value of the mpint. This call mechanism allows the routines better control over management of memory associated with the word pointers.

Function call	Effect
intMinus(&t, n)	$t = -n$
intSum(&t, n, m)	$t = n + m$
intDiff(&t, n, m)	$t = n - m$
intProd(&t, n, m)	$t = n * m$
intSqr(&t, n)	$t = n^2$
intExp(&t, n, m)	$t = n^m$
intDiv(&t, &s, n, m)	$t = n/m, s = n \,\%\, m$
intModProd(&t, n, m, s)	$t = nm \bmod s$
intModExp(&t, n, m, s)	$t = n^m \bmod s$
intPP(&t)	$t = t + 1$
intMM(&t)	$t = t - 1$
intInc(&t, a)	$t = t + a$
intDec(&t, a)	$t = t - a$
intSqrt(&t, n)	$t = \lfloor \sqrt{n} \rfloor$
intGCD(&t, n, m)	$t = \gcd(n, m)$
intBGCD(&t, n, m)	$t = \gcd(n, m)$ (Binary GCD)
intEGCD(&t, &r, &s, n, m)	$t = \gcd(n, m) = rn + sm$ (Extended GCD)

The above table should give sufficient information on how to use the built-in arithmetic routines. We need not elaborate them further. Instead we mention few topics not immediate from the table. Each of these functions returns nothing (that is, void). The function intDiv computes the quotient and remainder of an integer division simultaneously. If one is interested in only one of them, one may pass the NULL pointer as the other argument.[9] intDiv does not allow passing the same pointer for both these arguments. The routine intSqr(&t, n) is essentially equivalent to but more efficient than intProd(&t, n, n) and should be prefered to the latter when it is known *a priori* that both the numbers to be multiplied are the same.

Multiplying or dividing a multi-precision integer by another can be coded very efficiently when the second operand is a power of 2. Special GFL routines take care of these situations. These routines find extensive use for fields of characteristic 2. We list all these routines (and some other related ones) in the next table.

[9] One may pass NULL for both of them, but why should one do that, after all?

Function call	Effect
intTwo(n)	Returns 1 if $n = 2$, 0 otherwise
twoPowerToInt(&t, a)	$t = 2^a$
logTwo(n)	Returns $\lfloor \log_2(n) \rfloor$
intProdTwo(&t, n, a)	$t = n2^a$
intExpTwo(&t, n, a)	$t = n^{2^a}$
intDivTwo(&t, &s, n, a)	$t = n/2^a, s = n\%2^a$

As in the case of intDiv, one can supply the NULL pointer as the first or the second argument (or both!) to intDivTwo.

17.5 Prime numbers

Prime numbers are very important for many algorithms on finite fields. MPGFL provides facilities for checking a number for primality and for generating prime numbers. The routine prime(n) checks if the input mpint n holds a prime number. The test is probabilistic and is based on Miller-Rabin's pseudo-primality testing algorithm [20, 22]. Whenever the routine certifies a number to be non-prime (composite or unity or zero), it is always correct. On the other hand, when it certifies a number to be prime, there remains a small probability that the number is composite. The parameter that governs this probability of failure is #define'd in int.h as PRIME_ITER_CNT. This stands for the maximum number of bases over which the pseudo-primality of the input number is tested. The running time of prime is directly proportional to its value, whereas the probability of a false certificate (in the case mentioned above) is equal to the reciprocal of 4 raised to this value. int.h sets PRIME_ITER_CNT at the default value of 20. This means that there is a chance of 1 in 10^{12} when prime declares a composite number as prime. If one chooses, one may change this value. For example, the probability of failure can be reduced further by increasing PRIME_ITER_CNT at the cost of lower running speed. On the other hand, if speed is more important than reducing the possibility of error, one should decrease the value of PRIME_ITER_CNT. Whenever this value is changed, one must remember two things. One must recompile mpgfl for this change to be effective. Secondly, PRIME_ITER_CNT may not be assigned a value larger than 30.

The first prime generator we discuss is randPrime. It takes three arguments. The first is a pointer to an mpint to store the return value. The second is an int specifying the (exact) bit-length of the output mpint. The third is a short int that provides a directive on how to seed the random number generator. It has the same meaning as discussed in connection with randInt. In fact, randPrime repeatedly calls randInt to get a random integer of the desired bit-length and then checks for its primality by prime. It stops as soon as it listens a 'yes' from prime. randPrime returns a value – of type int – that indicates the number of integers checked for primality before (and including) the first (allegedly) prime number is found.

The other prime number generator – nextPrime – generates the nearest odd prime larger than or equal to a given mpint. As expected, it takes two arguments. The first one is the usual pointer to an mpint for storing the return value. The second one is an mpint telling nextPrime where to start searching for prime numbers. More specifically, nextPrime checks odd numbers larger than or equal to this second argument one by one in increasing order of magnitude until it gets a prime as certified by prime. Unlike randPrime, nextPrime returns nothing. The number of iterations that nextPrime goes through, can be easily calculated from the input and output values.

17.6 Integer factorization

The header file int.h defines a new data structure for holding the partial or complete factorization of a number. This is yet another struct with three components. The first one is an int that stores the number of distinct factors of an integer. The next two components are pointers to two dynamic arrays: the first one holds the multiplicities of the factors and the second one holds the factors themselves.

```
typedef struct {
    int nf;                 /* number of factors */
    int *multiplicity;      /* pointer to the array holding the multiplicities of the factors */
    mpint *factor;          /* pointer to the array of factors */
} intFactor;
```

As with other structures having pointers to dynamic arrays, an intFactor must be initialized before use. Similarly, there is a built-in routine that *destroys* an intFactor, that is, frees memory associated with an intFactor. The relevant library calls are

```
intFactor newIntFactor ();
void destroyIntFactor ( intFactor * );
```

In order to print the information stored in an intFactor, one should call the routine

```
int printIntFactors ( intFactor , FILE * );
```

The second argument passed to printIntFactors is a FILE pointer that has been opened in the write or append mode. This pointer may be NULL, in which case the output will be redirected to stdout. The final routine we describe in this context is

```
void factorizeInt ( intFactor * , mpint );
```

which computes the complete factorization of an integer and stores the same in the intFactor pointed to by the first argument.

17.7 Single-precision integer functions

The single-precision GFL also defines a set of routines for working with single-precision numbers, that is, data of type long int. These are the ones that are not available in the run-time library of C. We have already described these routines in connection with multi-precision integers. Here we will describe the syntax of these library calls that are particular to the single-precision library. Note that in the single-precision library an integer (long int) is not a struct with dynamic arrays. As a result, the single-precision GFL routines are allowed to return integers. One need not pass pointers to integers to effect assignments, as we did in connection with the multi-precision library. The prototypes of the library calls are, therefore, different from those of the calls that we have seen in the last few subsections. The header file where these routines are declared, is, again, int.h (more precisely, <gfl/int.h>).

long intExp (long n , long e) /* Returns n^e */
long intModExp (long n , long e , long m) /* Returns $n^e (\mathrm{mod} m)$ */
long intSqrt (long n) /* Returns $\lfloor \sqrt{n} \rfloor$ */
long intGCD (long n , long m) /* Returns $\gcd(n, m)$ */
long intBGCD (long n , long m) /* Returns $\gcd(n, m)$ (Binary GCD) */
long intEGCD (long *u , long *v , long n , long m)
/* Returns $\gcd(n, m) = un + vm$ (Extended GCD) */
/* where u and v are the values of u* and v* respectively */

```
int prime ( long n )                       /* Returns 1 if n is prime, 0 otherwise */
long randPrime ( int l , int seedInfo )    /* Returns a random prime of bit-length l. */
long nextPrime ( long n )                  /* Returns the nearest odd prime >= n */
```

The structure intFactor is defined as follows:

```
typedef struct {        /* structure holding the partial or complete factorization of a number */
    int nf;             /* number of factors */
    int *multiplicity;  /* pointer to the array holding the multiplicities of the factors */
    long *factor;       /* pointer to the array of factors */
} intFactor;
```

The four routines for the manipulation of intFactor have prototypes similar to those in the multi-precision library. That is,

```
intFactor newIntFactor ();                  /* Initialize intFactor */
void destroyIntFactor ( intFactor * );      /* Free memory associated with an intFactor */
int printIntFactors ( intFactor , FILE * ); /* Print the factors */
void factorizeInt ( intFactor * , long );   /* Compute the complete factorization of an integer */
```

18 The multi-precision Galois Field Library

In the last section, we have seen how we can represent and work with arbitrarily long signed multi-precision integers. We also termed the version of GFL that deals with these multi-precision numbers as "the multi-precision Galois Field Library" or "MPGFL" in short. Since the representation of numbers in MPGFL is different from that in the single-precision library, the GFL data types and routines should also be different in the two libraries. First of all, a GFelement is now no longer a long int, instead it is a synonym for mpint.

```
typedef mpint GFelement;
```

Most of the other data structures (namely, poly, matrix, vector etc.) that we have described in earlier sections retain the same description, since they are defined in terms of GFelement. Secondly, for efficient memory management, GFL routines never *return* a GFelement. An assignment is effected by passing to a routine a pointer to the mpint where we want to store the desired result. For example, we have seen that the single-precision library call

```
c = GFprod(a, b, K);
```

returns to c the long int obtained by multiplying GFelements a and b over the field K. For the multi-precision library, a, b and c are mpints and so the syntax for the call is

```
GFprod(&c, a, b, K);
```

One should note that the name of the function remains the same in the MPGFL (GFprod). This is true for *all* the functions described in Sections 3 through 15. The prototypes of the calls (i.e. the arguments and the return values) are different in those cases where the multi-precision library is expected to return an integer.[10] The most important consequence of this is that one can *not* include header files from both

[10] Every good rule has exceptions!

the libraries simultaneously. Depending on the cardinalities of the fields one wants to work with, one should choose *a priori* the library to include. As we pointed out earlier, it is always advisable to use the single-precision library whenever possible.

In this section, we describe all the data structures defined in MPGFL. We also list the exact syntax of all library calls that are particular to the multi-precision library. Since we have already described the functionality of these calls in details in connection with the single-precision library, we choose to be rather succinct here.

18.1 Data Types

The multi-precision library defines the following data types. We list these data structures under the respective header files where these are defined.

18.1.1 field.h

```
typedef mpint GFelement;

typedef short GF_d;

struct {
    GFelement p;          /* characteristic */
    GFelement q;          /* cardinality */
    int extlev;           /* Extension level */
    GF_d extof;           /* Extension of */
    long extdeg;          /* Extension degree (over immediate subfield) */
    long totextdeg;       /* Total extension degree (over prime subfield) */
    GFelement *defpoly;   /* Pointer to coefficients of defining polynomial */
    long *primpower;      /* Table of powers of a primitive element */
    long *disclog;        /* Table of discrete logs with respect to a primitive element */
    long *zechlt;         /* Zech's logarithm table */
} GF_info[MAX_FIELDS];

typedef struct {
    long degree;          /* The exact degree */
    GFelement *coeff;     /* Pointer to the array of coefficients */
} poly;
```

18.1.2 matrix.h

```
typedef struct {            /* Data structure matrix */
    int row;                /* Number of rows */
    int col;                /* Number of columns */
    GFelement **element;    /* Pointer to 2-dimensional array of matrix elements */
} matrix;
```

```
typedef struct {                /* Data structure vector */
    int size;                   /* Vector size */
    GFelement *element;         /* Pointer to the array of vector elements */
} vector;
```

18.1.3 factor.h

```
typedef struct {            /* Structure holding the partial or complete factorization of a polynomial */
    long nf;                /* Number of factors */
    long *multiplicity;     /* Pointer to the array holding the multiplicities of the factors */
    poly *factor;           /* Pointer to the array of factors */
} polyFactor;
```

18.1.4 dlog.h

```
/* Factor base for prime fields */
typedef struct {
    GFelement p;            /* characteristic of the field in which discrete log is taken */
    GFelement gen;          /* base to which discrete log is taken */
    int fbsize;             /* number of primes in the factor base */
    long *base;             /* elements of factor base */
    GFelement *baselog;     /* discrete logs of factor base elements */
} factorBase1;

/* Factor base for non-prime fields of extension level = 1 */
typedef struct {
    GFelement p;            /* characteristic of the field in which discrete log is taken */
    GFelement gen;          /* base to which discrete log is taken */
    int maxdeg;             /* maximum degree of irreducible polynomials in the factor base */
    int fbsize;             /* number of irreducible polynomials in the factor base */
    int nprime;             /* number of primes between 2 and p-1 */
    long *base;             /* elements of factor base */
    GFelement *baselog;     /* discrete logs of factor base elements */
} factorBase2;

/* Factor base for non-prime fields of extension level > 1 */
typedef struct {
    GFelement p;            /* characteristic of the field in which discrete log is taken */
    matrix ctop;            /* matrix for composed-to-polynomial basis transformation */
    GFelement gen;          /* base to which discrete log is taken */
    int maxdeg;             /* maximum degree of irreducible polynomials in the factor base */
    int fbsize;             /* number of irreducible polynomials in the factor base */
    int nprime;             /* number of primes between 2 and p-1 */
    long *base;             /* elements of factor base */
    GFelement *baselog;     /* discrete logs of factor base elements */
} factorBase3;
```

18.2 Subroutines

In this section we provide an alphabetic (case-insensitive) listing of all the subroutines declared in the MPGFL header files. All of these have been explained in details in Sections 3 through 16 in connection with the single-precision library. Here we will see the variants of these routines as applicable to the multi-precision library. For each routine we specify the exact prototype, a short description on what it does, and the header file where the routine is defined. For a more detailed description of the functionality of the routines, see the previous sections. This section highlights the prototypes of the library calls for MPGFL. We do not include the integer functions in this list – a detailed explanation of all these functions can be found in Section 17.

`void absNorm ( GFelement * , GFelement , GF_d )` Absolute norm – norm of a field element over the prime subfield `(field.h)`

`void absTrace ( GFelement * , GFelement , GF_d )` Absolute trace – trace of a field element over the prime subfield `(field.h)`

`void atoaTransMatrix ( matrix * , vector , vector , GF_d , GF_d )` transformation matrix from a general (arbitrary) basis to another general (arbitrary) basis `(basis.h)`

`int basis ( vector , GF_d , GF_d )` checks if the elements of the vector form a basis of the first field over the second `(basis.h)`

`void cardinality ( mpint * , GF_d )` number of elements in a Galois field `(field.h)`

`void characteristic ( mpint * , GF_d )` characteristic of a Galois field `(field.h)`

`void charPoly ( poly * , matrix , GF_d )` characteristic polynomial of a matrix `(polyutil.h)`

`void compMatrix ( matrix * , poly , GF_d )` companion matrix of a polynomial `(polyutil.h)`

`void copyMatrix ( matrix * , matrix )` copy a matrix to another `(matrix.h)`

`void copyPoly ( poly * , poly )` copy a polynomial to another `(poly.h)`

`void copyVector ( vector * , vector )` copy a vector to another `(matrix.h)`

`GF_d createExtGF ( GF_d , poly )` create the extension of a Galois field defined by the irreducible polynomial supplied as the second argument `(field.h)`

`void createFactorBase1 ( factorBase1 * , GF_d , GFelement , long )`
`void createFactorBase2 ( factorBase2 * , GF_d , GFelement , int )`
`void createFactorBase3 ( factorBase3 * , GF_d , GFelement , int )`
create a factor base (this includes computing discrete logarithms of the elements of the factor base) `(dlog.h)`

`GF_d createPrimeGF ( mpint )` create prime Galois field `(field.h)`

`void createPrimTable ( GFelement , GF_d )` create tables of powers of a primitive element `(primitive.h)`

`void createZechTable ( GF_d )` create Zech's logarithm table `(primitive.h)`

`void ctogTransMatrix ( matrix * , vector , GF_d , GF_d )` transformation matrix from composed to general basis `(basis.h)`

`void ctonTransMatrix ( matrix * , GFelement , GF_d , GF_d )` transformation matrix from the composed basis to a normal basis `(basis.h)` void cvtom (matrix * , vector) convert a column vector to a matrix (matrix.h)

void defPoly (GF_d , poly *) the defining polynomial of a Galois field over its 'immediate' subfield (field.h)

void destroyFactorBase1 (factorBase1 *)

void destroyFactorBase2 (factorBase2 *)

void destroyFactorBase3 (factorBase3 *)

free memory associated with a factorBase (dlog.h)

void destroyMatrix (matrix *) free memory associated with a matrix (matrix.h)

void destroyPoly (poly *) free memory associated with the coefficient array of a polynomial (poly.h)

void destroyPolyFactor (polyFactor *) free memory associated with a polyFactor (factor.h)

void destroyPrimTable (GF_d) free memory associated with primitive tables (primitive.h)

void destroyVector (vector *) free memory associated with a vector (matrix.h)

void destroyZechTable (GF_d) free memory associated with Zech's logarithm table (primitive.h)

void distinctDegreeFactorization (polyFactor * , poly , GF_d) compute the distinct degree factorization of a polynomial (factor.h)

void dlog1 (mpint * , GFelement , GF_d , factorBase1)

void dlog2 (mpint * , GFelement , GF_d , factorBase2)

void dlog3 (mpint * , GFelement , GF_d , factorBase3)

compute discrete log of a field element with the help of a factorBase (dlog.h)

void equalDegreeFactorization (polyFactor * , poly , int , GF_d) compute the equal degree factorization of a polynomial (factor.h)

int equalMatrix (matrix , matrix) check if two matrices are equal (matrix.h)

int equalPoly (poly , poly) check if two polynomials are equal (poly.h)

int equalVector (vector , vector) check if two vectors are equal (matrix.h)

void evalPoly (GFelement * , poly , GFelement , GF_d) evaluate a polynomial at a point (poly.h)

long extDeg (GF_d , GF_d) extension degree of the first field over the second (field.h)

int extLevel (GF_d) level of extension of a Galois field over its prime subfield (field.h)

void factorizePoly (polyFactor * , poly , GF_d) compute the complete factorization of a polynomial (factor.h)

void findFirstIrrPoly (poly * , GF_d , long) find the lexicographically first irreducible polynomial of given degree over the given field and store the result in the first argument (irr.h)

void findIsoMatrix (matrix * , GF_d , GF_d) find the isomorphism matrix between two finite fields of the same cardinality (iso.h)

void findLeastAffineMultiple (poly * , poly , GF_d , GF_d) find the least affine multiple of a polynomial (affine.h)

void findNormalElement (GFelement *, GF_d , GF_d , short) find a normal element in a field over a subfield (normal.h)

void findNormalElementA (GFelement *, GF_d , GF_d , short) find a normal element using the algorithm that follows from Artin's lemma (normal.h) void findNormalElementBDS (GFelement *, GF_d , GF_d) find a normal element using Bach, Driscoll and Shallit's algorithm (normal.h)

void findNormalElementR (GFelement *, GF_d , GF_d , short) find a random normal element (normal.h)

void findPrimElement (GFelement * , GF_d , short) find a random primitive element over the specified field. The last argument is a flag that dictates how to seed the random number generator: 1 – usr current time as seed, 2 – use the value of PRIMITIVE_SEED_VAL as seed, anything else - don't seed the random number generator at all (primitive.h)

void findPrimElement2 (GFelement * , GF_d , intFactor , short) same as findPrimElement except that the factorization of $q - 1$ is supplied as argument, where q is the cardinality of the field in which a primitive element is found (primitive.h)

int findRandomIrrPoly (poly * , GF_d , long , short) find a random irreducible polynomial of given degree over the given field and store the result in the first argument. The last argument specifies how to generate the seed for the random number generator. It can take the following values: 0 – Use the default seed, 1 – Use the local time as seed, 2 – Use the value of SEED_VAL as seed. The function returns the number of iterations that was necessary to find the irreducible polynomial (irr.h)

int findRoot (vector * , poly , GF_d , short) call one of the algorithms for finding roots of a polynomial in a given field. The choice of the algorithm is determined by the last argument. The actions corresp. to the different values are: 1 – Call findRootES, 2 – call findRootBR, 3 – call findRootBT, 4 – call findRootAM, Anything else – call one of these algorithms based on a fixed criterion (root.h)

void findRootAffinePoly (matrix * , vector * , vector * , poly , GF_d , GF_d) Find the roots of an affine polynomial (affine.h)

int findRootAM (vector * , poly , GF_d , GF_d) Berlekamp-Rumsey-Solomon algorithm that uses affine multiples (root.h)

int findRootBR (vector * , poly , GF_d) Berlekamp-Rabin algorithm – works only for fields of odd characteristic (root.h)

int findRootBT (vector * , poly , GF_d , GF_d) Berlekamp trace algorithm – suitable for fields of small characteristic and large extension degree (root.h)

int findRootES (vector * , poly , GF_d) exhaustive search algorithm (root.h)

void FrobeniusOrder (poly * , GFelement , GF_d , GF_d) compute the order of an element in the first field with respect to the Frobenius automorphism of the first field over the second. The order (a poly) is stored in the first argument (normal.h)

void GFdiff (GFelement * , GFelement , GFelement , GF_d) difference of two field elements (field.h)

void GFexp (GFelement * , GFelement , mpint , GF_d) a field elemenet raised to a power given by the mpint (field.h)

void GFinv (GFelement * , GFelement , GF_d) inverse of a field element (field.h)

void GFprod (GFelement * , GFelement , GFelement , GF_d) product of two field elements (field.h)

void GFqt (GFelement * , GFelement , GFelement , GF_d) quotient of two field elements (field.h)

void GFsum (GFelement * , GFelement , GFelement , GF_d) sum of two field elements (field.h) void gtocTransMatrix (matrix * , vector , GF_d , GF_d) transformation matrix from general to composed basis (basis.h)

void gtogTransMatrix (matrix * , vector , vector , GF_d) transformation matrix from a general basis to another general basis (for immediate extensions) (basis.h)

void gtopTransMatrix (matrix * , vector , GF_d) transformation matrix from a general basis to polynomial basis (for immediate extensions) (basis.h)

int identityMatrix (matrix) check if a matrix is the identity matrix (matrix.h)

int irreducible (poly , GF_d) check if the input polynomial is irreducible (irr.h)

void lc (GFelement * , poly) leading coefficient of a polynomial (poly.h)

int linIndep (vector * , int , GF_d) check if a set of vectors is linearly independent (matrix.h)

int linIndepCols (vector * , matrix , GF_d) find a set of linearly independent columns of a matrix (matrix.h)

int linIndepRows (vector * , matrix , GF_d) find a set of linearly independent rows of a matrix (matrix.h)

void linSysSolve (vector * , matrix , vector , GF_d) solve a system of linear equations (matrix.h)

void listAllIrrPoly (mpint * , GF_d , long , FILE * , short) list all irreducible polynomials of the given degree over the given field. The last argument specifies how one wants to format the coefficients of the polynomials. The function stores the number of irreducible polynomials found in the first argument. (irr.h)

void listAllNormalElements (GF_d , GF_d , FILE * , short) list all normal elements in a field over a subfield. (normal.h)

void listAllNPolys (GF_d , long , FILE * , short) list all N polynomials of given degree over the given field. (normal.h)

int LUPD (matrix * , vector * , matrix , GF_d) compute the LUP decomposition of a matrix (matrix.h)

void matrixDet (GFelement * , matrix , GF_d) determinant of a matrix (matrix.h)

void matrixDiff (matrix * , matrix , matrix , GF_d) difference of two matrices (matrix.h)

void matrixExp (matrix * , matrix , mpint , GF_d) raise a matrix to an integral power (matrix.h)

void matrixInv (matrix * , matrix , GF_d) inverse of a matrix (matrix.h)

void matrixProd (matrix * , matrix , matrix , GF_d) product of two matrices (matrix.h)

int matrixRank (matrix , GF_d) rank of a matrix (matrix.h)

void matrixSum (matrix * , matrix , matrix , GF_d) sum of two matrices (matrix.h)

void matrixTranspose (matrix * , matrix) transpose a matrix (matrix.h)

void matrixVectorProd (vector * , matrix , vector , GF_d) product of a matrix and a column vector (matrix.h)

void mctov (vector * , matrix , int) convert a specified column of a matrix to a vector (matrix.h)

void minimalPoly (poly * , GFelement , GF_d , GF_d) minimal polynomial of an element in the first field over the second (polyutil.h)

int monic (poly) check if the input polynomial is monic (poly.h) void monicize (poly * , GF_d) make the polynomial monic (poly.h)

void mrtov (vector * , matrix , int) convert a specified row of a matrix to a vector (matrix.h)

void mtocv (vector * , matrix) convert a matrix to a column vector (matrix.h)

void mtorv (vector * , matrix) convert a matrix to a row vector (matrix.h)

factorBase1 newFactorBase1 ()
factorBase2 newFactorBase2 ()
factorBase3 newFactorBase3 ()
initialize a factorBase (dlog.h)

matrix newMatrix ( ) initialize a matrix (matrix.h)

poly newPoly ( ) initialize a poly (poly.h)

polyFactor newPolyFactor () initialize a polyFactor (factor.h)

vector newVector ( ) initialize a vector (matrix.h)

void norm (GFelement * , GFelement , GF_d , GF_d) Norm of a field element over the specified subfield (field.h)

int normal (GFelement , GF_d , GF_d) check if a field element is normal over a subfield (normal.h)

int NPoly (poly , GF_d) check if a polynomial is an N-polynomial (normal.h)

void ntocTransMatrix (matrix * , GFelement , GF_d , GF_d) transformation matrix from a normal basis to the composed basis (basis.h)

void ntopTransMatrix (matrix * , GFelement , GF_d) transformation matrix from a normal basis to polynomial basis (for immediate extensions) (basis.h)

void overspLinSysSolve (vector * , matrix , vector , GF_d) solve an overspecified system of linear equations (matrix.h)

int permPoly (poly , GF_d) checks if the input polynomial is a permutation polynomial over the finite field supplied as the second argument (permutation.h)

void polyBasis (vector * , poly * , GF_d , GF_d , short) polynomial basis of a field over a subfield (basis.h)

void polyDerivative (poly * , poly , GF_d) formal derivative of a polynomial (polyutil.h)

void polyDiff (poly * , poly , poly , GF_d) difference of two polynomials (poly.h)

void polyDisc (GFelement * , poly , GF_d) discriminant of a polynomial (polyutil.h)

void polyDiv (poly * , poly * , poly , poly , GF_d) quotient and remainder of division of two polynomials (poly.h)

void polyEGCD (poly * , poly * , poly * , poly , poly , GF_d) extended gcd of two polynomials (poly.h)

void polyExp (poly * , poly , mpint , GF_d) polynomial raised to an integral power (poly.h)

void polyFactorRefine (polyFactor * , polyFactor , GF_d) the general factor refinement algorithm (for any number of polynomials) (normal.h)

void polyGCD (poly * , poly , poly , GF_d) gcd of two polynomials (poly.h) void polyModExp (poly * , poly , poly , mpint , GF_d) modular exponentiation of a polynomial (poly.h)

void polyModProd (poly * , poly , poly , poly , GF_d) modular product of two polynomials (poly.h)

int polyPairRefine (polyFactor * , poly , poly , int , int , GF_d) compute the factor refinement for two polynomials. Returns true iff the second factor is reduced to a constant polynomial (normal.h)

void polyProd (poly * , poly , poly , GF_d) product of two polynomials (poly.h)

void polyRes (GFelement * , poly , poly , GF_d) Resultant of two polynomials (polyutil.h)

void polySum (poly * , poly , poly , GF_d) sum of two polynomials (poly.h)

GF_d primeSubGF (GF_d) the prime subfield of a Galois field (field.h)

int primitive (GFelement , GF_d) check if the GFelement is primitive over the field (primitive.h)

int primitive2 (GFelement , GF_d , intFactor) check if the GFelement is primitive over the field given the factorization of $q - 1$ where q is the cardinality of the finite field (primitive.h)

long printFactors (polyFactor , GF_d , FILE * , short) print the factors stored in a polyFactor (factor.h)

int printGFElement (GFelement , GF_d , FILE * , short) print element of GF in different formats. The last argument is a flag. Admissible values of this flag are : 0 – Single number, 1 – Vector over its immediate subfield, 2 – Vector of vectors of ... over prime subfield, 3 – Flattened form of 2, 4 – Same as 3 except without parentheses, 5 – Polynomial in last extending element, 6 – Polynomial in extending elements (default) (field.h)

void printGFInfo (GF_d) print information on Galois field (field.h)

void ptogTransMatrix (matrix * , vector , GF_d) transformation matrix from polynomial basis to a general basis (for immediate extensions) (basis.h)

void ptonTransMatrix (matrix * , GFelement , GF_d) transformation matrix from polynomial basis to a normal basis (for immediate extensions) (basis.h)

void readFactorBase1 (factorBase1 * , char *)

void readFactorBase2 (factorBase2 * , char *)

void readFactorBase3 (factorBase3 * , char *)

read factorBase from hard disk (dlog.h)

void readMatrix (matrix *) read a matrix from stdin (matrix.h)

void readMatrixFromArray (matrix * , int , int , GFelement **) read a matrix from a 2-d array (matrix.h)

void readPoly (poly *) read polynomial from stdin (poly.h)

void readPolyFromArray (poly * , long , GFelement *) read polynomial from an array (poly.h)

void readPrimTable (GF_d , char *) read primitive power tables from hard disk (primitive.h)

void readVector (vector *) read a vector from stdin (matrix.h)

void readVectorFromArray (vector * , int , GFelement *) read a vector from array (matrix.h)

void readZechTable (GF_d , char *) read Zech's logarithm table from hard disk (primitive.h)

void rvtom (matrix * , vector) convert a row vector to a matrix (matrix.h)

void saveFactorBase1 (factorBase1 , char *)
void saveFactorBase2 (factorBase2 , char *)
void saveFactorBase3 (factorBase3 , char *)
save factorBase in hard disk for future reuse (dlog.h)

void savePrimTable (GF_d , char *) save primitive power tables in hard disk files for future use (primitive.h)

void saveZechTable (GF_d , char *) save Zech's logarithm table in a hard disk file for future use (primitive.h)

void scalarMatrixProd (matrix * , GFelement , matrix , GF_d) product of a matrix with a scalar (matrix.h)

void scalarVectorProd (vector * , GFelement , vector , GF_d) product of a vector with a scalar (matrix.h)

void setIdentityMatrix (matrix * , int) set a square matrix to the identity matrix (matrix.h)

void setZeroMatrix (matrix * , int , int) set a matrix to the zero matrix (matrix.h)

void setZeroVector (vector * , int) set a vector to the zero vector (matrix.h)

int singular (matrix , GF_d) check if the input matrix is singular (matrix.h)

int sqLinSysSolve (matrix *, vector * , vector * , matrix , vector , GF_d) solve a square system of linear equations (matrix.h)

void squareFreeFactorization (polyFactor * , poly , GF_d) compute the square free factorization of a polynomial (factor.h)

GF_d subGF (GF_d) subfield of which the argument is an (immediate) extension (field.h)

int symmetricMatrix (matrix) check if a matrix is symmetric (matrix.h)

long totExtDeg (GF_d) extension degree of the input field over its prime subfield (field.h)

void trace (GFelement * , GFelement , GF_d , GF_d) trace of a field element over the specified subfield (field.h)

void underspLinSysSolve (matrix *, vector * , vector * , matrix , vector , GF_d) solve an underspecified system of linear equations (matrix.h)

void vectorDiff (vector * , vector , vector , GF_d) difference of two vectors (matrix.h)

void vectorMatrixProd (vector * , vector , matrix , GF_d) product of a row vector and a matrix (matrix.h)

void vectorSum (vector * , vector , vector , GF_d) sum of two vectors (matrix.h)

void vtomc (matrix , vector , int) set the specified column of a matrix from a vector (matrix.h)

void vtomr (matrix , vector , int) set the specified row of a matrix from a vector (matrix.h)

void writeMatrix (matrix , GF_d , FILE * , short) write a matrix to stdout (matrix.h)

int writePoly (poly , GF_d , FILE * , short) print a polynomial – The third argument specifies the mode of printing the coefficients. This mode is the same as in printGFElement. It returns the number of characters printed (poly.h)

void writeVector (vector , GF_d , FILE * , short) write a vector to stdout (matrix.h)

int zeroMatrix (matrix) check if a matrix is the zero matrix (matrix.h) int zeroPoly (poly)   check if a polynomial is the zero polynomial (poly.h)

int zeroVector (vector)   check if a vector is the zero vector (matrix.h)

19 Some examples

So far we have seen how one can represent various algebraic entities using GFL. We have also described various routines for manipulation of these data structures. We now demonstrate how we can write programs that use these data structures and library calls. We explain the major steps that the programmers should follow while developing their own application programs using GFL.

19.1 Example 1: Computation of characteristic roots of a matrix

In this example, we explain the programming techniques for the single-precision library. We write a small code for the computation of the characteristic roots (i.e. eigen values) of a matrix with proper multiplicities.

Include header files

One should first include GFL header files to tell the C compiler about the new data structures and external procedure declarations defined in GFL. One may choose only the individual GFL header files that are needed for the particular program, or all header files. For our example, we include the following

```
#include <stdio.h>          /* C standard io header file */
#include <gfl/int.h>        /* Integer manipulation routines */
#include <gfl/field.h>      /* Field arithmetic */
#include <gfl/poly.h>       /* Polynomial arithmetic */
#include <gfl/matrix.h>     /* Matrix arithmetic */
#include <gfl/polyutil.h>   /* Some polynomial utilities */
```

An easier way to include the necessary files is to include all files – both the ones we need in our program and the ones that we do not.

```
#include <stdio.h>          /* C standard io header file */
#include <gfl/all.h>        /* Include all GFL header files */
```

Initialize GFL

This is a very important step. Any program that uses GFL must do this before doing anything else. This step carries out certain book-keeping tasks and sets up some tables for later use. If the library is not initialized, one would get bizarre results like unwelcome halts, nasty segmentation faults and so on. Initialization of the GFL kit is rather easy. One should just call

```
GFLinitialize();
```

Declare variables

We first decide what data we need to represent. We then declare them using the user-defined data types introduced in the last section. In our example, we need a few field descriptors (of type GF_d) for referring to various fields, a matrix whose characteristic roots will be calculated, a poly to hold the characteristic polynomial of this matrix, a vector to store the roots of this polynomial, and some other auxiliary variables of type long.

```
GF_d F, K, L;          /* The field descriptors */
poly f;                /* Polynomials */
matrix M;              /* Matrices */
vector v;              /* Vectors */
long i, n;             /* Auxiliary variables */
```

Initialize variables

Any GFL structure that has dynamic arrays must first be initialized before they can be used. In this example, the variables f, M, v contain dynamic arrays. These arrays are initialized to NULL as follows:

```
f = newPoly(); M = newMatrix(); v = newVector();
```

Alternatively one may explicitly make the pointers in these structures NULL as

```
f.coeff = NULL; M.element = NULL; v.element = NULL;
```

We emphasize that whenever a new variable with dynamic arrays is declared (as global variables, inside procedures or loops, or even inside another variable, e.g. in a structure containing polynomials or matrices etc.), one must first initialize the variable before anything is done with the variable. Many GFL routines free the memory associated with dynamic arrays (unless they are NULL) before they are assigned new memory. An uninitialized non-NULL value can, therefore, lead to run-time hazards.

Create fields

Let's say that we want to compute the characteristic values of a matrix over $\mathbb{F}_{3^4}$ in the extension field $\mathbb{F}_{3^{12}}$. To this end, we first create the prime field of characteristic 3. Then we find a random irreducible polynomial of degree 4 over $\mathbb{F}_3$ and attach a root of this polynomial to $\mathbb{F}_3$ in order to get the extension field $\mathbb{F}_{3^4}$. In a similar fashion we extend $\mathbb{F}_{3^4}$ by an irreducible polynomial of degree 3 over $\mathbb{F}_{3^4}$. This gives us $\mathbb{F}_{3^{12}}$. All these can be done very simply by a few library calls.

```
F = createPrimeGF(3);          /* Create the prime field of characteristic 3 */
findRandomIrrPoly(&f, F, 4, 1);  /* Find a random irreducible polynomial of degree 4 over F */
K = createExtGF(F, f);         /* Extend F by f */
findRandomIrrPoly(&f, K, 3, 1);  /* Find a random irreducible polynomial of degree 3 over K */
L = createExtGF(K, f);         /* Extend K by f */
```

Each call to createPrimeGF or createExtGF returns a field descriptor (GF_d) that we shall use for all future references to the respective fields. We have already explained in Section 6 that the last argument of findRandomIrrPoly is a directive to the random polynomial generator routine on how to seed the random number generator. findRandomIrrPoly returns the total number of polynomials checked before (and including) the first irreducible polynomial is found. We choose to ignore this value here.

Do computations

We are now ready to carry out the actual computations. First we have to assign a matrix to the variable M. There are many ways in which this can be done. For the time being, we read it from stdin in an interactive fashion.

```
readMatrix(&M);
```

This asks the user for the numbers of rows and columns of M and then the elements in the row-major order. We intend to let M store a matrix over K, i.e., $\mathbb{F}_{3^4}$. As explained in the previous section, the elements of $\mathbb{F}_{3^4}$ are represented as numbers between 0 and $3^4 - 1 = 80$ (both inclusive). So we must input a value in this range for every element of M. Since readMatrix does not know in advance how we are going to interpret these values, it does not complain if we supply element values not in the above range. The responsibility of entering meaningful values is, therefore, on the user.

In the next step we compute the characteristic polynomial of M. This is also easy. GFL provides the built-in procedure charPoly to do this.

```
charPoly(&f, M, K);
```

We then find all the roots of f in the extension L. We call another built-in routine findRoot to do this.

```
n = findRoot(&v, f, L, 0);
```

The roots are stored as elements in the vector v. The number of roots can be found from v.size or the value returned by findRoot (assigned to n above). The last argument to findRoot tells findRoot which algorithm to use (See Section 8 for details).

Now we have all the characteristic roots of M in the extension field L. What is left is to compute the multiplicities of these characteristic roots. One can do it in several ways. For example, instead of computing the roots of f, we can factorize f and read the multiplicities from those of the linear factors of f. Another possibility is to divide f successively by $x - a$ for each characteristic root a of f till a non-zero remainder is found. We follow the second approach, because it illustrates how one can do arithmetic on polynomials. We need three auxiliary polynomials g, rem and quot.

```
poly g, rem, quot;

/* Initialize new variables */
g.coeff = rem.coeff = quot.coeff = NULL;

/* set g to be a polynomial of degree 1 */
g.degree = 1;
g.coeff = (GFelement *)malloc(2 * sizeof(GFelement));
g.coeff[1] = 1;

/* loop for each characteristic root of f */
for (i=0; i<n; i++) {
    int mul;
```

```
    /* Set the constant term of g to –a, where a is the ith characteristic root of f */
    /* Note that –a = (–1) * a, and (–1) has the representation p–1 in F, K and L */

    g.coeff[0] = GFprod(v.element[i], p – 1, L);

    /* GFprod returns the product of its first two arguments considered as elements
       of the field supplied as the third argument */

    mul = 0;        /* Initialize multiplicity to 0 */

    do {
        polyDiv(&quot, &rem, f, g, L);
            /* Divide f by g and store the quotient in quot and remainder in rem */
            /* L is the field where the coefficient arithmetic takes place */

        if (zeroPoly(rem)) {
            /* If the remainder is zero, that is, if (rem.degree == MINUS_INFINITY) */
            mul++;        /* Increase multiplicity by 1 */
            copyPoly(&f, quot);        /* Store the quotient in f */
        }

    } while (zeroPoly(rem));

    printf("Multiplicity of %ld is %d\n", v.element[i], mul);
}
```

The above example clearly illustrates how easy it is to do the desired task by simple library calls.

Wind up

Now that our program has printed the characteristic roots of the input matrix together with their multiplicities, we may choose to exit from the program. A better approach is first to free the memory allocated to the `coeff` arrays of `poly`, the `element` arrays of `matrix` and `vector` and so on. These can be done by explicit calls to `free`. Alternatively, the following routines can be used.

```
destroyPoly(&f);   destroyPoly(&g);   destroyPoly(&quot);   destroyPoly(&rem);
destroyMatrix(&M);   destroyVector(&v);
```

In general, it is always a good practice to free the memory allocated to dynamic arrays whenever the contents of the memory are no longer needed. This need not be done only at the very end of the programs. In our example, we might call `destroyMatrix(&M)` immediately after the characteristic polynomial of `M` is calculated. We don't need the elements of `M` after this step.

Compilation

Let's say that we have saved the code for our example as described above in a file named eigen.c.[11] Now we compile eigen.c as follows:

```
cc eigen.c -o eigen -lgfl
```

This creates the executable eigen. Note that one must link the GFL static library by the -l option during compilation. If one wants to optimize the code, one might use the -O flag.

Now we should type eigen at the command prompt for this program to run.

19.2 Example 2: ElGamal's public key cryptosystem

Let us illustrate the multi-precision library calls with another example. We develop a finite field cryptosystem proposed by ElGamal [9]. We implement three basic subroutines for the cryptosystem using MPGFL calls. These subroutines do not constitute a complete program. They demonstrate how one can write subroutines in one's program using MPGFL. Before we write the codes, we state some general remarks about MPGFL.

- The MPGFL header files can be included as

  ```
  #include <mpgfl/all.h>
  ```

  or as

  ```
  #include <GFL/all.h>
  ```

- One must perform the initialization step in any program that include MPGFL. The process is exactly the same as we mentioned in relation to the single-precision library in the previous example.

  ```
  GFLinitialize();
  ```

- One must initialize mpint and GFelement in the MPGFL. One need not do that in the single-precision library. The initialization routine for mpint is called newInt().
- The multi-precision library can be linked during compilation by the directive -lmpgfl or -lGFL, for example, as

  ```
  cc source.c -lGFL
  ```

  Note that it is not possible to include both the single-precision and the multi-precision libraries together. That is, one must not specify both -lgfl and -lGFL during compilation. Similarly one must not #include files simultaneously under gfl and GFL.

Back to our example. Suppose that A (Alice) wants to send a message to B (Bob) over a channel where a third party C (Carol) may intercept A's messages and read the secret information meant for B only. To befool C, A and B choose a large finite field K ($\mathbb{F}_{2^{401}}$ or $\mathbb{F}_{2^{503}}$, for example) and compute a

[11] Actually more! One must put everything (except the #include statements and optionally the variable declarations) in the main function. We didn't mention that. A C programmer should know all these!

primitive element g in that field. B then selects a random integer b and computes $l = g^b$. B publishes the public key l and keeps the private key b a secret. This process of key generation can be implemented as follows.

```
void makekey ( GFelement *b , GFelement *l , GFelement g , GF_d K ) {
    mpint qmo;

    qmo = newInt();          /* Initialize qmo */
    cardinality(&qmo, K);    /* Store in qmo the cardinality of K */
    intMM(&qmo);             /* qmo -- */
    randRes(b, qmo);         /* Store at *b a random non-negative number less than qmo */
    GFexp(l, g, *b, K);      /* Calculate l = g^b in K */
    destroyInt(&qmo);        /* Return to system the memory held by qmo */
}
```

Note that in an actual implementation, one should discard trivial values of b, say 0 or 1. We do not indicate this step here for the sake of simplicity. Note also how the temporary mpint qmo is initialized, used and then destroyed in the routine. After B calls this subroutine as

```
makekey(&b, &l, g, K);
```

the mpint b holds B's private key and the mpint l holds his public key.

Now A wants to send an element $m \in K$ to B. She carries out the following three steps to encrypt her message.

1. choose a random integer t and compute $e_1 = g^t$, where $0 < t < q - 1$, q = the cardinality of K.
2. read B's public key (l) and compute $e_2 = ml^t$.
3. send the pair (e_1, e_2) to B.

A can use the following subroutine for the purpose of encryption.

```
void encrypt ( GFelement *e1 , GFelement *e2 , GFelement g , GFelement m ,
               GFelement l , GF_d K ) {
    GFelement t, qmo;

    t = newInt();   qmo = newInt();            /* Initialize mpint */
    cardinality(&qmo, K);   intMM(&qmo);       /* Set qmo to q - 1 */
    do   randRes(&t, qmo);   while (zeroInt(t));
                              /* Generate a random integer between 1 and q - 2 */
    GFexp(e1, g, t, K);       /* The first part e1(= g^t) of encrypted message */
    GFexp(e2, l, t, K);       /* Compute l^t */
    GFprod(e2, *e2, m, K);    /* The second part e2(= ml^t) of encrypted message */
    destroyInt(&t);           /* Free memory */
    destroyInt(&qmo);         /* Free memory */
}
```

A now sends e_1 and e_2 to B. B recovers the message m from these values using his private key b in the following way. We have $e_1 = g^t$ and $e_2 = ml^t = mg^{bt}$, so that $m = e_2 \cdot g^{-bt} = e_2 \cdot e_1^{-b}$. The subroutine for decryption is, thus:

```
void decrypt ( GFelement *m , GFelement e1 , GFelement e2 , GFelement b , GF_d K ) {
    GFexp(m, e1, b, K);          /* Compute e_1^b */
    GFinv(m, e1, K);             /* Compute (e_1^b)^{-1} = e_1^{-b} */
    GFprod(m, *m, e2, K);        /* Compute e_2 · e_1^{-b} */
}
```

Since $e_1^{q-1} = 1$ in K if $e_1 \neq 0$, we can calculate e_1^{-b} efficiently as $e_1^{q-1-b} = e_1^{-b}$. We leave out the details of the implementation.

Now let's talk about C. She has knowledge of K, g and l, but not of b. Suppose C gets e_1 and e_2 and wants to decipher the original message m. She must first calculate b which is the discrete logarithm of l in K with respect to g. If she goes through the GFL manual carefully, she will be happy to see that GFL provides built-in routines for the computation of discrete logs in finite fields. Similar to the above procedures, it is very easy to write a routine for Carol, that uses these GFL library calls. But the bad news for Carol is that this algorithm does not run in time polynomially bounded by the size of K. Indeed it is *sub-exponential*, so that if K is sufficiently large (say, $\mathbb{F}_{2^{503}}$), C cannot compute discrete logarithms in K in feasible time. But C must not blame the designers of GFL – at present no better algorithms are known for computing discrete logs in finite fields. And this is why ElGamal's scheme of encryption is secure.

20 Debugging with GFL

The Galois Field Library provides subroutines that help one check whether variables of data types defined in GFL header files contain admissible or valid data. For example, if we want to use a GFelement c as an element of a field K of cardinality q, we must have $0 \leq c \leq q - 1$. We call any other value of c an inadmissible or invalid value. For a poly f, on the other hand, the checking procedure is more complicated. First of all, it checks whether each of its coefficients is valid and then whether f.degree reflects the exact degree of the polynomial etc. Similarly for matrix, vector, polyFactor and so on. The user should call the following subroutines to check the validity of data of these types. The header file where these checking routines are declared is called debug.h. Each of these routines returns 0, if we pass valid data to it. In case some argument contains invalid data, it prints out diagnostic messages regarding why it considers the data invalid, and then returns a non-zero value.

Prototype	**Effect**
int checkInt (mpint)	checks the validity of an mpint (applicable for MPGFL)
int checkIntFactor (intFactor)	checks the validity of an intFactor
int checkGF_d (GF_d)	checks the validity of a GF_d
int checkGF_info (GF_d)	checks the GF_info records for a GF_d
int checkGFelement (GFelement , GF_d)	checks whether a GFelement is a valid element of a GF_d
int checkPoly (poly , GF_d)	checks whether a poly is a valid polynomial over a field
int checkMatrix (matrix , GF_d)	checks whether a matrix is a valid matrix over a field

int checkVector (vector , GF_d)	checks whether a vector is a valid vector over a field
int checkPolyFactor (polyFactor , GF_d)	checks whether a polyFactor contains valid factorization info over a field
int checkFactorBase1 (factorBase1 , GF_d)	checks validity of a type 1 factor base
int checkFactorBase2 (factorBase2 , GF_d)	checks validity of a type 2 factor base
int checkFactorBase3 (factorBase3 , GF_d)	checks validity of a type 3 factor base

The most common run-time error that occurs when one runs a buggy program using GFL calls, is the *segmentation fault*. This occurs when one tries to make an invalid memory access (for example, when one tries to access data pointed to by a NULL pointer, or when one frees memory that has not been allocated). It is possible to write one's own interrupt service routine to deal with such situations. This can be done by the UNIX™ command signal. Whenever segmentation fault occurs, the signal SIGSEGV is raised. One has to write the corresponding interrupt handler and notify the run-time system to use this interrupt handler whenever SIGSEGV is raised. One may call the checking routines described in the previous table inside the signal handling routine. This is a useful debugging technique for programs that involve GFL data structures. We explain the method in a short example below.

```
/* Include the relevant C header file */
#include <signal.h>

/* Include GFL header files */
#include <gfl/field.h>
#include <gfl/poly.h>
#include <gfl/matrix.h>
...
#include <gfl/debug.h>

GF_d K;
GFelement a;
poly f;
matrix M;

/* Write SIGSEGV handler */
void sigsegvHandler ( int sig ) {

    /* Call the debugging routines for printing diagnostic information */
    checkGF_d(K);
    checkGF_info(K);
    checkGFelement(a, K);
    checkPoly(f, K);
    checkMatrix(M, K);

    fprintf(stderr, "Better luck next time ...\n");
    exit (-1);
}
```

```
. . .
int main() {
    /* Notify the C compiler of the SIGSEGV handler routine */
    signal(SIGSEGV, sigsegvHandler);

    GFLinitialize();
    . . .
    /* Do buggy computations */
    . . .
    exit(0);
}
```

21 Conclusion

In this report we provided a complete description of GFL, an algebraic computation library for working over finite fields. The comprehensive nature of GFL and careful implementation of algorithms make the tool readily usable for design and study of various application systems on finite fields. Readers should find this report useful to learn the paradigms that they should follow in order to write application programs involving GFL library calls.

There are many ways in which the library can be enhanced. We mention a few important possibilities.

1. *Improving performance of GFL routines:* This involves devising and/or implementing algorithms that run more efficiently compared to the routines currently implemented in GFL.
2. *Adding new features:* Facilities for manipulating multi-variate polynomials and polynomial functions can be added to GFL.
3. *Designing a front-end:* An interpreter that runs on top of GFL can make programmer's task much easier and user-friendly.
4. *Parallelization:* The GFL routines can be parallelized and run in a distributed fashion on a network of processors. One possible way to achieve this is to implement a client-server environment with the help of Unix™ domain sockets.

References

[1] L. Adleman, 'A subexponential algorithm for the discrete logarithm problem with applications to cryptography', *20th Annual Symp. on Foundations of Comp. Sc.* (1979), 55–60.

[2] E. Artin, 'Galois Theory', University of Notre Dame Press, South Bend, Ind., 1966.

[3] E. Bach, J. Driscoll and J. Shallit, 'Factor Refinement', *J. of Algorithms*, **15(2)** (1993), 199-222.

[4] E. R. Berlekamp, 'Factoring polynomials over large finite fields', *Math. Comp.*, **24** (1970), 713–735.

[5] E. R. Berlekamp, H. Rumsey and G. Solomon, 'On the solution of algebraic equations over finite fields', *Information and control*, **10** (1967), 553–564.

[6] P. Camion, 'Improving an algorithm for factoring polynomials over a finite field and constructing large irreducible polynomials', *IEEE trans. Info. Th.*, **29** (1983), 378–385.

[7] D. Cantor and H. Zassenhaus, 'A new algorithm for factoring polynomials over finite fields', *Math. Comp.*, **36** (1981), 587–592.

[8] A. Das and C. E. Veni Madhavan, 'GFL – A library for computing over Galois fields', Submitted to *Software–Practice and Experience*, 1998.

[9] T. ElGamal, 'A public key cryptosystem and a signature scheme based on discrete logarithms', *IEEE trans. Info. Th.*, **31**, 469–472 (1985).

[10] J. von zur Gathen and M. Giesbrecht, 'Constructing normal bases in finite fields', *J. Symbolic Computation*, **10** (1990), 547–570.

[11] I. N. Herstein, 'Topics in Algebra', 2nd. Edition, John Wiley & Sons, 1975.

[12] K. Hoffman and R. Kunze, 'Linear Algebra', 2nd. Edition, Prentice-Hall, 1971.

[13] K. Huber, 'Some comments on Zech's logarithms', *IEEE trans. Info. Th.*, **36** (1990), 946–950.

[14] B. W. Kernighan and D. M. Ritchie, 'The C Programming Language', 2nd edition, Prentice-Hall, 1988.

[15] H. W. Lenstra, 'Finding isomorphisms between finite fields', *Math. Comp.*, **56** (1991), 329–347.

[16] R. Lidl and H. Niederreiter, 'Finite fields', Cambridge University Press, 1987.

[17] K. McCurley, 'The discrete logarithm problem', *Cryptology and Computational Number Theory*, Proc. Symp. in Appl. Math., **42** (1990), 49–74.

[18] A. J. Menezes, ed., 'Applications of finite fields', Kluwer Academic Publishers, 1993.

[19] M. Mignotte, 'Mathematics for Computer Algebra', Springer-Verlag, 1992.

[20] G. Miller, 'Riemann's hypothesis and tests for primality', *J. Comput. and System Sc.*, **13** (1976), 300–317.

[21] A. Odlyzko, 'Discrete logarithms and their cryptographic significance', *Advances in Cryptology: Proc. of Eurocrypt '84*, Lecture Notes in Computer Science, **209** (1985), Springer-Verlag, 224–314.

[22] M. Rabin, 'Probabilistic algorithms for testing primality', *J. Number Theory*, **12** (1980), 128–138.

[23] I. E. Shparlinski, 'Computational problems in finite fields', Kluwer Academic Publishers, 1992.

[24] D. Wan, 'A p-adic lifting lemma and its applications to permutation polynomials', In *Proc. conf. finite fields and their applications*, vol. 141 of *Lecture notes in pure and applied mathematics*, Marcel Dekker, 1993, 209–216.

[25] A. E. Western and J. C. P. Miller, 'Tables of indices and primitive roots', Royal Mathematical Tables, vol. 9, Cambridge University Press, 1968.