Finite Fields Computations in Maxima

Fabrizio Caruso caruso@dm.unipi.it
Jacopo D'Aurizio elianto84@gmail.com
Alasdair McAndrew amca01@gmail.com

April, 2008

This file documents the Maxima file "gf.mac", which is a Maxima package for finite fields. This package is part of standard Maxima since version 5.14. If you use version 5.15 the package is split in two files gf.mac and gf_roots.mac. In subsequent versions the whole package is contained in gf.mac. This package is also suitable for teaching and exploration and it offers an easier interface than other comparable libraries such as the finite library in Axiom (provided by the domain constructor FFP). The first version of the package was based on the paper "Finite Fields Manipulations in Macsyma" by Kevin Rowley and Robert Silverman, SIGSAM 1989, but for which the source code is long gone. The version included in Maxima contains lots of new features and optimizations implemented by Fabrizio Caruso and Jacopo D'Aurizio.

In order to use the package it is enough to start and version of Maxima from 5.15 and use the command:

```
(%i2) load(gf);
```

will load all the functions. If you use version 5.14 then you must also do

```
(%i2) load(gf_roots);
```

Getting started

All user commands are prefixed with "gf_"; you need to start by entering the parameters for your field. All fields in this package are of the form

$$\mathbb{F}_p[x]/m(x)$$

where p is a prime number and $m(x)$ is an polynomial irreducible over $\mathbb{F}_p$. If the degree of $m(x)$ is n, the the finite field will contain p^n elements, each element being a polynomial of degree strictly less than n, and all coefficients being in $\{0, 1, \ldots, p-1\}$. Such a field is called a *finite field* or *Galois field* of order p^n, and is denoted $\mathbb{F}_{p^n}$. Note that although there are many different irreducible polynomials to choose from, if $m(x)$ and $n(x)$ are different polynomials irreducible over $\mathbb{F}_p$ and of the same degree, then the fields

$$\mathbb{F}_p[x]/m(x)$$

and

$$\mathbb{F}_p[x]/n(x)$$

are isomorphic.

In these fields, addition and subtraction are performed on the coefficients modulo p, and multiplication and division modulo $m(x)$.

To create a field, you need the parameters p, $m(x)$ and optionally n which is the degree of $m(x)$. The form of the command is

```
gf_set(p,n,m(x));
```

or

```
gf_set(p,m(x));
```

or

```
gf_set(p);
```

`gf_set` checks that p is prime, and if `GF_IRREDUCIBILITY_CHECK` is set to `true` it also checks whether $m(x)$ is irreducible. If these conditions are satisfied, the command performs some background calculations (discussed below), and returns true.

```
(%i3) gf_set(2,4,x^4+x+1);
(%o3)                          true
```

The package comes also with two test files `gf_test.mac` and `gf_hard_test.mac` which can be run with the command `batch(<path_to_test_file>,test)`.

Having set up the field, we can now perform arithmetic on field elements:

Addition/subtraction. These are performed with the commands "`gf_add`" and "`gf_sub`". In the particular field entered above, since all arithmetic of coefficients is performed modulo 2, addition and subtraction are equivalent:

```
(%i4) a:x^3+x^2+1;
(%o4)                      x^3 + x^2 + 1
(%i5) b:x^3+x+1;
(%o5)                       x^3 + x + 1
(%i6) gf_add(a,b);
(%o6)                         x^2 + x
```

Multiplication. This is performed with the command "`gf_mul`":

```
(%i7) gf_mul(a,b);
(%o7)                         x^2 + x
```

Inversion and division. The inverse of a field element $p(x)$ is the element $q(x)$ for which their product is equal to 1 (modulo $m(x)$). This is performed using "`gf_inv`". In a finite field, division is defined as multiplying by the inverse; thus

$$a(x)/b(x) = a(x)(b(x))^{-1}.$$

These operations are performed with the commands "`gf_inv`" and "`gf_div`":

```
(%i8) gf_inv(b);
```

(%o8) x^2+1

```
(%i9) gf_div(a,b);
```

(%o9) x^3+x^2

```
(%i10) gf_mul(a,gf_inv(b));
```

(%o10) x^3+x^2

Exponentiation. To raise a field element to an integer power, use "`gf_exp`":

```
(%i11) gf_exp(a,10);
```

(%o11) x^2+x+1

```
(%i12) gf_exp(a,15);
```

(%o12) 1

Random elements. Finally, a random element can be obtained with "`gf_rand()`":

```
(%i13) makelist(gf_rand(),i,1,4);
```

(%o13) $[x^3+x^2+x, x^3+x^2+x+1, x^3+x^2+x+1, x^2+x]$

```
(%i14) M:genmatrix(lambda([i,j],gf_rand()),3,3);
```

(%o14)
$$\begin{pmatrix} x^3+x^2+x & x^3 & x^3+x^2 \\ x^2 & x^3+x^2+1 & x^3+x+1 \\ x^2+x & x^3+x^2+x+1 & x^2 \end{pmatrix}$$

Primitive elements, powers and logarithms

The non-zero elements of a finite field form a multiplicative group; a generator of this group is a *primitive element* of the field. The command "`gf_findprim()`" finds a primitive element:

```
(%i15) gf_findprim();
(%o15)
```

$$x$$

Given that any non-zero element in the field can be expressed as a power of this primitive element, this power is the *index* of the element; its value is obtained with "`gf_ind`":

```
(%i16) a:x^3+x^2+1;
```

(%o16) x^3+x^2+1

```
(%i17) gf_ind(a);
```

(%o17) 13

```
(%i18) ev(a=gf_exp(x,13)),pred;
(%o18)                          true
```

Since every element of the field can be represented as a polynomial

$$a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_2x^2 + a_1x + a_0$$

where every coefficient a_i satisfies $0 \leq a_i \leq p-1$, a field element can also be considered as a list:

$$[a_{n-1}, a_{n-2}, \ldots, a_2, a_1, a_0].$$

This list can be considered as the "digits" of a number in base p, in which the field element is equivalent to the number

$$a_{n-1}p^{n-1} + a_{n-2}p^{n-2} + \cdots + a_2p^2 + a_1p + a_0.$$

Thus every polynomial is equivalent to a number between 0 and $p^n - 1$; this number can be obtained by the "`poly2num`" command. This command is actually not needed by the user; it is used for some internal commands. The other direction is given by "`num2poly`".

Since every non-zero field element $a = a(x)$ is both equivalent to a number A and a power i of a primitive element e, we can create a list of powers corresponding to particular numbers. This list, `gp`, which is created by `gf_set`, is defined as follows: its i-th element is the numerical form of the i-th power of the primitive element. Thus, if

$$a(x) \equiv A \equiv e^i$$

where e is the primitive element, then the i-th element of `gp` is A. By definition we have $e^{p^n-1} = 1$.

The numbers A run over all integers from 1 to $p^n - 1$, and the powers i run over all the integers from 0 to $p^n - 2$, there is a corresponding "logarithm" list, `lg`. The logarithm table may be considered to be indexed from 0 to $p^n - 2$, and its i-th element is the power corresponding to that element:

```
(%i19) lg[10];
(%o19)                          9
(%i20) num2poly(9);
(%o20)                        x^3 + x
(%i21) gf_ind(%);
(%o21)                          9
```

The creation of the lists `gp` and `lg` may take a few seconds, but they only have to be done once.

Logarithms. The list `lg` produces the logarithm (with respect to the primitive element `e`) of any non-zero element of the field. But the logarithm of any element relative to the base of another can be obtained with the command "`gf_log`":

```
(%i22) a:x^2+1+1;
(%o22)                      x^2 + x + 1
(%i23) b:x^3+x^2+1;
(%o23)                     x^3 + x^2 + 1
(%i24) gf_log(a,b);
(%o24)                          10
```

We conclude that, in our field, $a = b^{10}$.

Primitive elements. A given field will have many primitive elements, and the command "`gf_primep`" tests an element to see if it is primitive:

```
(%i25) gf_primep(x^3+x+1);
(%o25)                          true
(%i26) gf_primep(x^2+x);
(%o26)                          false
```

Order. By definition, any element a of the field will satisfy $a^{p^n-1} = 1$. The *order* of a is the *lowest* power m for which $a^m = 1$. It will be a factor of $p^n - 1$, and is obtained with "`gf_ord`":

```
(%i27) gf_ord(x^2+x);
(%o27)                            3
(%i28) gf_ord(x^3+x+1);
(%o28)                           15
```

Minimal polynomials

Associated with every element $a \in GF(p^n)$ is a polynomial $p(x)$ which satisfies:

1. $p(a) = 0$,
2. the coefficient of the highest power in $p(x)$ is one,
3. for any other polynomial $q(x)$ with $q(a) = 0$, $p(x)$ is a divisor of $q(x)$.

The polynomial $p(x)$ is thus, in a very strict sense, the *smallest* polynomial which has a as a root. It is the *minimal polynomial* of a. The command "`gf_minpoly`" calculates it:

```
(%i29) a:x^3+x+1$
(%i30) p:gf_minpoly(a);
(%o30)                      z^5 + z^3 + 1
```

To check this, substitute a for z in p:

```
(%i31) subst(a,z,p);
(%o31)           (x^3 + x + 1)^5 + (x^3 + x + 1)^3 + 1
(%i32) gf_eval();
(%o32)                            0
```

An application: the Chor-Rivest knapsack cryptosystem

The Chor-Rivest knapsack cryptosystem is the only knapsack cryptosystem which doesn't use modular arithmetic; instead it uses the arithmetic of finite fields. Although it has been broken, it is still a very good example of finite field arithmetic.

Assuming the two protagonists are Alice and Bob, and Alice wishes to set up a public key for Bob to encrypt messages to her. Alice chooses a finite field $\mathbb{F}_{p^n} = \mathbb{F}_p[x]/m(x)$, and a random primitive element $g(x)$. She then computes $a_i = \log_{g(x)}(x+i)$ for every $i \in \mathbb{F}_p$. She selects a random integer d for which $0 \le d \le p^n - 2$, and computes $c_i = (a_i + d) \pmod{p^n - 1}$. Her public key is the sequence c_i, with the parameters p and n.

To encrypt a message to Alice, Bob encodes the message as binary blocks of length p which contain n ones. Given one such block $M = (M_0, M_1, \ldots, M_{p-1})$, Bob creates the cipher text

$$c = \sum_{i=0}^{p-1} M_i c_i \pmod{p^n - 1}$$

which he send to Alice.

To decrypt c, Alice first computes $r = (c - nd) \pmod{p^n - 1}$, and then computes $u(x) = g(x)^r \pmod{m(x)}$. She then computes $s(x) = u(x) + m(x)$ and factors s into linear factors $x + t_i$. The t_i values are the positions of the ones in the message block M.

Actually, the complete cryptosystem also involves a permutation, which is applied to the sequence a_i to create c_i. But for this example we are just interested in the field arithmetic.

We shall choose the example given in chapter 8 of HAC, but without the permutation. Here the field is

$$GF(7^4) = \mathbb{F}_7[x]/(x^4 + 3x^3 + 5x^2 + 6x + 2)$$

and the primitive element chosen is $g(x) = 3x^3 + 3x^2 + 6$ and the random integer d is 1702.

First, Alice must compute her public key:

```
(%i33) gf_set(7,4,x^4+3*x^3+5*x^2+6*x+2);
(%o33)                          true
(%i34) g:3*x^3+3*x^2+6;
(%o34)                     3x^3 + 3x^2 + 6
(%i35) gf_primep(g);
(%o35)                          true
(%i36) a:makelist(gf_log(x+i,g),i,0,6);
(%o36)              [1028, 1935, 2054, 1008, 379, 1780, 223]
(%i37) d:1702$
(%i38) c:makelist(mod(a[i]+d,base^exp-1),i,1,7);
(%o38)              [330, 1237, 1356, 310, 2081, 1082, 1925]
```

Now Bob can encrypt a message to Alice; suppose one such block is $[1, 0, 1, 1, 0, 0, 1]$, which is a block of length 7 which contains exactly 4 ones.

```
(%i39) M:[1,0,1,1,0,0,1];
(%o39)                     [1, 0, 1, 1, 0, 0, 1]
(%i40) ord:base^exp-1;
(%o40)                          2400
```

```
(%i41) c:mod(sum(M[i]*c[i],i,1,7),ord);
(%o41)                              1521
```

This last value is the ciphertext. Alice now needs to decrypt it:

```
(%i42) r:mod(c-exp*d,ord);
(%o42)                              1913
(%i43) u:gf_exp(g,r);
(%o43)                     $x^3+3x^2+2x+5$
(%i44) s:u+px;
(%o44)                  $x^4+4x^3+8x^2+8x+7$
(%i45) factor(s);
(%o45)                  $(x-1)x(x+2)(x+3)$
```

The t_i values are thus $-1, 0, 2, 3$, which modulo 7 are $6, 0, 2, 3$—and these are the positions of the ones in M.

Matrices

There are two commands for dealing with matrices over finite fields: "`gf_matinv`" for inverting a matrix, and "`gf_matmul`" for multiplying matrices. Using the matrix `M` generated previously:

```
(%i46) MI:gf_matinv(M);
(%o46)
```

$$\begin{pmatrix} x^2 & x^3+x^2+x+1 & x^3 \\ x^3+x+1 & x^2+x & x^3+x+1 \\ x & 0 & x^3+x+1 \end{pmatrix}$$

```
(%i47) gf_matmul(M,MI);
(%o47)
```

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Normal Bases

Any field $GF(p^n)$ may be considered as a vector space over $\mathbb{F}_p$; one basis is the set

$$\{1, x, x^2, \ldots, x^{n-1}\}$$

which is called the *polynomial basis*. A *normal element* is a field element e for which the set

$$\{e, e^p, e^{p^2}, \ldots, e^{p^{n-1}}\}$$

forms a basis. There are several commands for dealing with normal elements and bases. The command "`gf_findnorm()`" finds a normal element by simply picking field elements at random and testing each one for normality. Although this is a probabilistic algorithm, in practice it works very quickly:

```
(%i48) gf_set(2,10,x^10+x^3+1);
(%o48)                              true
(%i49) pe:gf_findnorm();
(%o49)
```

```
                    x^9 + x^7 + x^6 + x^5 + x^4 + x^2 + x
```

The command “`gf_sfindnorm()`” is a brute force search through all field elements; in general it is slower than `gf_findnorm()`.

Having found a normal element the command “`gf_normbasis()`” produces a matrix the rows of which are the coefficients of the basis elements e^{p^k}. This command takes an optional parameter; a polynomial p. If present, `gf_normbasis()` checks if the field element is normal, and if so, produces the matrix, otherwise prints an error message. If the parameter is not given, `gf_normbasis()` first finds a normal element, and then uses that element to produce the matrix.

With the normal basis, the command “`gf_nbrep(p,M)`” produces the normal basis representation of `p`, with respect to the basis `M`, as a list of coefficients. One attraction of using normal bases is that much arithmetic can be simplified; for example, in a normal basis representation, a power of the prime p is equivalent to a shift of coefficients:

```
(%i50) M:gf_normbasis(pe)$
(%i51) a:gf_rand();
(%o51)              x^9 + x^7 + x^6 + x^5 + x^3 + x^2 + x
(%i52) gf_nbrep(a,M);
(%o52)                   [0, 0, 0, 1, 0, 1, 0, 0, 1, 0]
(%i53) gf_nbrep(gf_exp(a,2),M);
(%o53)                   [0, 0, 1, 0, 1, 0, 0, 1, 0, 0]
```

Large fields

If the flag `largefield` is set to `false` then the `gf_set` command will not find a primitive element, instead it will compute a full table of powers and logarithms. If the flag is set to `true`, the opposite happens: no tables are precomputed and a primitive element is found. The default value for `largefield` is `true` because this is the best choice for large and medium size finite fields.

The default value of `largefield` has the following effects

1. A primitive element is found, but the power and logarithm tables are not calculated.
2. The command `gf_exp`, instead of using table lookups, uses the simple “repeat squaring” algorithm.
3. The command `gf_log`, instead of using table lookups, uses the Pohlig-Hellman algorithm. This is most efficient if $p^n - 1$ has only small prime factors.

A disadvantage is that every operation now may take more time when the finite field is extremely small.

```
(%i54) largefield:true$
(%i55) gf_set(2,20,x^20+x^3+1);
(%o55)                         true
(%i56) a:x^15+x^5+1;
(%o56)                     x^15 + x^5 + 1
(%i57) gf_ind(a);
(%o57)                        720548
```

```
(%i58) gf_exp(a,3^12);
(%o58)                x^17 + x^16 + x^13 + x^12 + x^11 + x^3 + x^2 + x
```

It is up to the user to decide when a field is large enough for the `largefield` flag to be set to `false`.

Square and cube roots

Multiple algorithms have been implemented in order to solve the square and cube root extraction problem over $\mathbb{F}_p$; all of them basically perform an exponentiation in a extension field (ie $\mathbb{F}_{p^2} = \mathbb{F}_p[x]/(x^2+bx+a)$ or $\mathbb{F}_{p^3} = \mathbb{F}_p[x]/(x^3-bx-a)$) through a repeated-squaring scheme, reaching so a complexity of $O(n\log(p))$ multiplications in $\mathbb{F}_p$; however, due to some differences in the representation and multiplication of elements in the extension field, they do not have exactly the same running time:

1. `msqrt(a,p)` returns the two square roots of a in $\mathbb{F}_p$ (if they exist) representing every k-th power of x in $\mathbb{F}_p[x]/(x^2+bx+a)$ as the first column of the matrix M^k, where M is the companion matrix associated with the polynomial x^2+bx+a and b^2-4a is a quadratic non-residue in $\mathbb{F}_p^*$. It requires $5\log_2(p)$ multiplications in $\mathbb{F}_p$.
2. `ssqrt(a,p)` returns the two square roots of a in $\mathbb{F}_p$ (if they exist) using Shanks algorithm. It requires $5\log_2(p)$ multiplications in $\mathbb{F}_p$.
3. `gf_sqrt(a,p)` returns the two square roots of a in $\mathbb{F}_p$ (if they exist) using the Muller algorithm (an improved, shifted version of Cipolla-Lehmer's) and should reach the best performance, requiring only $2\log_2(p)$ multiplications in $\mathbb{F}_p$.
4. `mcbrt(a,p)` returns the cube roots of a in $\mathbb{F}_p$ (if they exist) representing every k-th power of x in $\mathbb{F}_p[x]/(x^3+bx+a)$ as the vector $(M_{2,2}, M_{2,3}, M_{3,2})$ in the matrix M^k, where M is the companion matrix associated with the polynomial x^3+bx+a, irreducible over $\mathbb{F}_p$ (Stickelberger-Redei irreducibility test for cubic polynomials is used). It requires $10\log_2(p)$ multiplications in $\mathbb{F}_p$.
5. `scbrt(a,p)` follows the same multiplication steps of `mcbrt(a,p)`, using a simpler polynomial representation for the elements of the field extension. It requires about $11\log_2(p)$ multiplications in $\mathbb{F}_p$.
6. `gf_cbrt(a,p)` returns the cube roots of a in $\mathbb{F}_p$ (if they exist) using the generalized Shanks algorithm: it's pretty fast, requiring about $4\log_2(p)$ multiplications in $\mathbb{F}_p$, being so the candidate choice for cube root extraction.

Other implemented routines, using about the same ideas, are:

1. `fastfib(n)` and `fastlucas(n)`, returning the n-th Fibonacci and Lucas numbers through a Muller-like scheme; they require exactly 2 squarings and 3 sums for each bit in the binary representation of n, having so a bit-complexity bounded by $2\log_2(n)^{3+\varepsilon}$, with ε depending on the adopted integer squaring algorithm.
2. `qsplit(p)` and `csplit(p)`, splitting a prime p over $\mathbb{Z}[i]$ and $\mathbb{Z}[\omega]$, ie finding (a,b) such that $p = a^2+b^2$ (this is possible only when p is in the form $4k+1$) or $p = a^2+ab+b^2$ (this is possible only when p is in the form $3k+1$), by the reduction of a binary quadratic form of the proper discriminant. They have the same complexity of the computation of a single Jacobi symbol, $O(\log(p)^2)$ bit-operations.

```
(%i8) msqrt(64,1789); ssqrt(64,1789); gf_sqrt(64,1789);
(%o8)                          [1781, 8]
(%o9)                          [8, 1781]
(%o10)                         [1781, 8]
(%i11) mcbrt(64,1789); scbrt(64,1789); gf_cbrt(64,1789);
(%o11)                       [4, 608, 1177]
(%o12)                       [4, 608, 1177]
(%o13)                       [4, 1177, 608]
(%i14) modulus:1789; factor(x^3-64); modulus:false;
(%o14)                            true
(%o15)                 (x - 608)(x - 4)(x + 612)
(%o16)                           false
(%i17) qsplit(1789); csplit(1789);
(%o17)                           [5, 42]
(%o18)                          [12, 35]
(%i19) fastfib(137); fastlucas(141);
(%o19)             19134702400093278081449423917
(%o20)            293263001536128903730947142076
```