An Introduction to Computational Tools for Error Correction Coding

Todd K. Moon
Utah State University
January 20, 2005

1 Introduction

The computations occurring in error correction coding — dealing with finite fields and polynomials — are frequently laborious to deal with by hand. The programming tools which are developed in the laboratory exercises associated with the book can help. However, they require programming and compiling specialized code. What would be more helpful is a sort of finite field/polynomial desk calculator. Even better would be a programmable desk calculator. Fortunately, tools that provide interactive computational capability exist. In this note, we discuss two of them. The first is called `gap`, the second is called `magma`.

These tutorials introduce explicitly several useful capabilities. They also provide examples of other useful capability, such as `if` statements, `for` loops, printing, and so forth.

2 gap

Gap is a system for computational discrete algebra, with libraries containing thousands of functions. It can do *much* more than is necessary as a computational aid for this course! It is also freely available. See `www.gap-system.org`. In this introduction we summarize only the points which are most pertinent to computations in error correction coding, leaving untouched most of the vast capability of this tool.

Syntactically, all commands entered in `gap` must end with a semicolon `;`, and `gap` will print the result. To suppress printing of a result, use two semicolons `;;`. Assignments are done using the `:=` operation. Scroll-back and editing commands are available using arrow keys or the emacs-like bindings (C-p for previous line, C-f for forward character, etc.). A help system is accessed using `?`.

This tutorial barely scratches the surface of the capabilities. Reference to the manual or online help is strongly suggested. Parts of this tutorial material is drawn from material posted at the website above.

2.1 Finite Fields and Polynomial Rings over Finite Fields

To create a finite field in `gap` use a statement such as

```
F := GF(5);
```

To create an indeterminate to be used with polynomials over this field, use

```
x := Indeterminate(F,"x");
```

Then polynomials can be entered just as one might expect:

```
p := 3*x^2 + 4*x;
```

However, `gap` echoes this response with

```
Z(5)^3*x^2 - x;
```

showing some peculiarities of the operation. The number `Z(5)` is a primitive element in the field. `gap` is implicitly informing you that `Z(5)`^{3} is equal to the field element 3. To convert back to the integer representation in this case, use the function `IntFFE` (Integer form of a Finite Field Element):

```
IntFFE(Z(5)^3);
```

to which `gap` responds

```
3
```

(It is this portrayal of field objects in terms of primitive elements that I personally find distracting, and why I prefer `magma`.) To find out what primitive element `gap` is using, try

```
IntFFE(Z(5));
```

In gap, the integers are not considered to be in the field. Using the gap command in (to check for set membership) we can test:

```
0 in F;
```

to which gap responds false. We can create a variable which represents the 0 and 1 element (the additive and multiplicative identities) using special gap commands:

```
zero := Zero(F);
one := One(F);
```

to which gap responds

```
0*Z(5);
Z(5)^0;
```

Computations in $GF(2)[x]$ are especially easy.

```
F := GF(2);
x := Indeterminate(F,"x");
```

Then polynomial operations are straightforward. For example

```
(x^2 + x+1)*(x^2 + x^5 + x);
(x^2 + x+1) + (x^2 + x^5 + x);
(x^2 + x+1) - (x^2 + x^5 + x);
```

Consider now some examples over $GF(2^4)$.

```
F:= GF( 2^4 );
one:= One(F);
Elements( F );
```

Now the prime field of F should be $GF(2)$ and F should have degree 4 over $GF(2)$

```
PrimeField( F );
DegreeOverPrimeField( F );
```

The defining polynomial in this case is

```
g := DefiningPolynomial( F );
```

which turns out to be $x^4 + x^2 + 1$ which is, of course, is irreducible.

```
Factors(DefiningPolynomial( F ));
```

Its root is called in gap Z(24):

```
RootOfDefiningPolynomial( F );
```

For computational purposes, it is convenient to call this thing a (for α).

```
a := RootOfDefiningPolynomial(F);
```

Then and of course Z(24) satisfies this irreducible polynomial:

```
a^4 + a + 1;
```

gives the result 0*Z(2). You could also evaluate the polynomial $g(\alpha)$ as

```
Value(g,a);
```

If you want to build a field using your own irreducible polynomial you could do something like the following:

```
x:= Indeterminate(GF(2), "x");      ??
FF:= GF( 2, 1 + x^3 + x^4 );
```

Basic polynomial operations are straightforward:

```
(a^4*x^3 + a^2*x^2 + a^5*x + 1) * (a^8*x^2 + a^7*x + 1);
(a^4*x^3 + a^2*x^2 + a^5*x + 1) - (a^8*x^2 + a^7*x + 1);
```

Computing quotients (division without remainder) and remainders uses the following commands

```
EuclideanQuotien(a^4*x^3 + a^2*x^2 + a^5*x + 1,a^8*x^2 + a^7*x + 1);
EuclideanRemainder(a^4*x^3 + a^2*x^2 + a^5*x + 1,a^8*x^2 + a^7*x + 1);
```

The GCD is accessed using `Gcd`. The coefficients in the GCD are found using `GcdRepresentation`.

Factors can be found using `Factors`:

```
Factors(p1);
```

2.2 Vectors and Matrices in GAP

We begin by looking at ways to define a vector.

```
v1:= [ 1/2, 3/4, -2/3, 22/7 ];
```

We can see if what we have defined is a vector

```
IsVector( v1 );
```

We define a vector space over the rational numbers as Suppose we want a three dimensional subspace of R4

```
v2:= [ 1, 3, 2, 4 ];
v3:= [ 1/2, 1/4, 1/3, 3/4 ];

V:= VectorSpace( Rationals, [ v1, v2, v3 ] );
```

It is easy to test if vectors lie in the vector space V

```
[ 1, 1, 1, 1 ] in V;
[ 28, 70, 84, 45 ] in V;
```

Scalar multiplication of vectors is easy

```
1/2*v1;
```

as is the dot product

```
v1*v2;
```

Of course linear combinations of vectors from V lie in V. We try an example

```
1/2*v1 - 3/4*v2 + 5/4*v3 in V;
```

Vectors over a finite field: let us construct the 3 dimensional vector space over GF(3)

```
V:= FullRowSpace( GF( 3 ), 3 );
```

Let us check whether a vector is in V

```
[ 1, 1, 1 ] in V;
```

This gives false since 1 is not in $GF(3)$ in the `gap` notation. We need to use the identity of GF(3)

```
o:= One( GF( 3 ) );
[ 1, 1, 1 ]*o in V;
```

Bases We set up a vector space over the rationals spanned by 4 vectors

```
v1:= [ 2, 2, 1, 3 ];;
v2:= [ 7, 5, 5, 5 ];;
v3:= [ 3, 2, 2, 1 ];;
v4:= [ 2, 1, 2, 1 ];;
V:= VectorSpace( Rationals, [ v1, v2, v3, v4 ] );
```

We compute its basis

```
B:= Basis( V );
```

What is the dimension of V?

```
Length( B );
```

Let us see the basis vectors GAP has computed

```
BasisVectors( B );
```

We can express vectors in V as a linear combination of the basis vectors B

```
Coefficients( B, [2,1,2,1]);
```

Since the coefficients are $[2, -1, 1/4]$ we should have

```
[ 2, 1, 2, 1 ] = 2*B[1] - 1*B[2] + 1/4*B[3];
```

What if we try to express a vector not in the space as a linear combination?

```
Coefficients( B, [1, 0, 0, 0]);
```

Not surprisingly we get "fail".

We can take linear combinations of the vectors of B with the `LinearCombination` command. For example

```
LinearCombination( B, [ 1/2, 1/3, 1/4 ] );
```

produces the same result as

```
1/2*B[1] + 1/3*B[2] + 1/4*B[3];
```

Defining matrices.

The following command defines a matrix as a list of lists.

```
m1:= [ [ 1, 2, 3 ], [ 2, 3, -1 ], [ 1, -2, 5] ];
```

We can display this on the screen as a more standard looking matrix.

```
Display( m1 );
```

To print the (2, 3)-entry of `m1` we use

```
m1[2][3];
```

To change an entry use

```
m1[2][2]:=300;
Display( m1 );
```

Next we define a matrix over GF(5) We let `o` be the identity of GF(5) and `z` be the zero of GF(5).

```
o:= One( GF( 5 ) );
z:= Zero( GF( 5 ) );
```

Then build the matrix

```
m2:= [ [ o, z, z ], [ 2*o, 3*o, z ], [ z, o, 4*o ] ];
```

Now `Display` gives a more readable form

```
Display( m2 );
```

We can easily define matrices whose (i, j)th entry is given by a function of i and j.

```
m3:= List( [ 1, 2, 3 ], i -> List( [ 1, 2, 3 ], j -> i*j ) );
Display(m3);
```

Of course it is easy to define large matrices this way

```
m4:= List( [ 1 .. 12 ], i -> List( [ 1 .. 12 ], j -> i*j ) );
Display( m4 );
```

Defining the same matrix, but this time over Z(5) can be done by either

```
m5:= List( [ 1 .. 12 ], i -> List( [ 1 .. 12 ], j -> i*j*o ) );

m5:= List( [1 .. 12], i -> List( [1 .. 12], j -> i*j ) )*o;
```

The identity matrix must be square.

```
id:= IdentityMat( 5 );;
Display( id );
```

The zero matrix can be rectangular

```
zm:= NullMat(6, 8);;
Display( zm );

zm:= NullMat(6, 7, GF(2));;
Display( zm );      ??
```

A diagonal matrix is square. We give it the diagonal entries.

```
dm:= DiagonalMat( [ 1, 1, 2, 2, 3, 4 ] );;
Display( dm );
```

We can define a similar matrix over a finite field.

```
o:= One(GF( 3 ));;
dm:= DiagonalMat( [ 1, 1, 2, 2, 3, 4 ]*o );;
Display( dm );
```

Matrix algebra
Addition, multiplication, etc. for matrices is straightforward. We give examples

```
m:= [1..8];;
m[1]:= [[1,2,3], [2,3,-1],[ 1,-2,5]];;
m[2]:= [[-1,4,-3], [1,2,-1],[ -1,-2,3]];;
m[3]:= m[1] + m[2];
m[4]:= m[1]*m[2];
m[5]:= 3*m[1] - 7*m[2];
m[6]:= m[1]^3;
```

```
m[7]:= m[2]^(-1);
m[8]:= m[6]*(4*m[5] + 6*m[2]*m[4])^(-1);

for i in [1..8] do
  print(m[i]);
  print("\n");
od;
```

To find the transpose of a matrix use `TransposedMat`.

```
m:=[ [1,2,3], [2,3,-1], [1,-2,5] ];;
mdash:=TransposedMat(m);
Display(m);
Display(mdash);
```

Some determinants We use the **Determinant** function in the obvious way

```
m1:= [ [1,2,3], [2,3,-1], [1,-2,5] ];;
m2:= [ [-1,4,-3], [1,2,-1], [-1,-2,3] ];;
Determinant( m1 );
Determinant( m2 );
```

If we look at matrices over GF(2) then their determinants will be either 0 or 1. What proportion would one expect for each with 2 2 matrices? Let us examine the situation.

```
o:= One( GF(2) );;
z:= Zero( GF(2) );;

countOne:= 0;;
countZero:= 0;;
for i in [1..1000] do
  mat:= RandomMat(2, 2, GF(2));
   if Determinant(mat) = o then
     countOne:= countOne + 1;
   else
     countZero:= countZero + 1;
   fi;
od;
Print(countOne, " ", countZero, "\n");      ??
```

Obviously more matrices have determinant 0 than have determinant 1. Can you see why this is so?

What happens with 3 3 matrices, 4 4 matrices, 5 5 matrices?

```
countOne:= 0;;
countZero:= 0;;
for i in [1..1000] do
  mat:=RandomMat(5, 5, GF(2));
    if Determinant(mat) = o then
      countOne:= countOne + 1;
    else
      countZero:= countZero + 1;
    fi;
od;
Print(countOne, " ", countZero, "\n");      ??
```

Eigenvalues and eigenvectors To compute the eigenvalues and eigenvectors of a matrix we need to specify the field

```
m:= [1..3];;
m[1]:= [ [1,2,3], [2,3,-1], [1,-2,5] ];;
m[2]:= [ [-1,4,-3], [1,2,-1], [-1,-2,3] ];;
m[3]:= [ [-2,-3,-3], [-1,0,-1], [5,5,6] ];;
for i in [1,2,3] do
  Print(Eigenvalues(Rationals, m[i]), "\n");
  Print(Eigenvectors(Rationals, m[i]), "\n");
  Print(CharacteristicPolynomial(m[i]), "\n");
  Print("\n");
od;
```

Let us do the same calculation over GF(7)

```
o:= One( GF(7) );;
m:= [1..3];;
m[1]:= [ [1,2,3], [2,3,-1], [1,-2,5] ]*o;;
m[2]:= [ [-1,4,-3], [1,2,-1], [-1,-2,3] ]*o;;
m[3]:= [ [-2,-3,-3], [-1,0,-1], [5,5,6] ]*o;;
for i in [1, 2, 3] do
  Print(Eigenvalues(GF(7), m[i]), "\n");
  Print(Eigenvectors(GF(7), m[i]), "\n");
  Print(CharacteristicPolynomial(m[i]), "\n");
  Print("\n");
od;
```

Systems of linear equations

If we are given a system of linear equations XA = B, where B is a row vector, then we can find a solution to the equations (of course only if one exists).

```
A:= [[1,2,1],[1,-1,2],[1,2,1]];
B:=[1,1,4/3];

SolutionMat(A,B);
```

Now this only gives one solution, even when the system of equations has infinitely many solutions. The general solution is then SolutionMat(A, B) + v for any v in NullspaceMat(A). Let us check that fact for our particular example.

```
NullspaceMat(A);
```

Since the null space has dimension 1 the general solution will be the particular solution SolutionMat(A, B) plus any multiple of the basis vector for NullspaceMat(A). For example

```
M:= SolutionMat(A,B) + 5/11*NullspaceMat(A)[1];
M*A = B;
```

In this example the system of equations has a solution. We could have checked this by seeing that the rank of A was equal to the rank of the augmented matrix

```
Rank(A);
```

To create the augmented matrix of the system we take a copy of A, then add the vector B as the final row.

```
Add(augA,B);
augA:= ShallowCopy(A);
```

Now augA and A should have the same rank since our system had a solution.

```
Rank(augA) = Rank(A);
```

Of course if the rank of the augmented matrix is different from the rank of A then the system has no solution. Here is an example.

```
B:=[1,-1,3,8];       ??
augA:= ShallowCopy(A);       ??
Add(augA,B);
Rank(augA) = Rank(A);       ??
```

If we tried to solve XA = B, for this different row vector B, then we will find no solution to the equations and SolutionMat(A, B) will return fail.

To create spaces for matrices, the following commands are useful

```
M := RMatrixSpace(GF(5),2,3);
  // create the space of all 2x3 matrices with coefficients in GF(5)
M := MatrixRing(GF(5),3);
  // create the space of all 3x3 matrices with coefficients in GF(5)
```

Matrices over polynomial rings

We begin by defining R to be the ring of polynomials over the rational numbers in the indeterminate x. First we set up a list of names that GAP will use for the indeterminates; in this case it only contains one entry.

```
indetnames:= ["x"];
```

Now define R to be the polynomial ring in a single indeterminate which GAP will write as we have specified.

```
R:= PolynomialRing(Rationals, indetnames);
```

Now we want to use "x" for the name of the indeterminate so we set this up.

```
indets:= IndeterminatesOfPolynomialRing(R);
x:= indets[1];
```

We let o be the identity of R.

```
o:= One( R );
```

We must be careful when we define matrices over R to make sure we use the notation 2*o for the integer 2 in R. The element 2 itself is an integer, but not an element of R. Let us define a matrix over R.

```
mat:= [[o, x],[2*o + x, x^2]];
```

To check that mat is a matrix use the IsMatrix command.

```
IsMatrix( mat );
```

Note that the following will not define a matrix as the elements do not come from a ring (remember that 1 and 2 are not in R but x is in R).

```
mat1:= [ [1, x], [2+x, x^2] ];
IsMatrix(mat1);
```

We can carry out the usual matrix operations with mat.

```
mat^2;
mat^(-1);
```

To convert `mat1` to a matrix we could multiply it by `o`.

```
mat1:= mat1*o;
IsMatrix(mat1);
mat1^(-1);
Determinant(mat1);
```

Determinants of matrices over polynomial rings

We set up a polynomial ring over the rationals with 4 indeterminates. As before we first set up the names that GAP will use for the indeterminates, then set up the names that we can use for the indeterminates (which for convenience we make the same).

```
indetnames:= ["t","x","y","z"];;
R:= PolynomialRing(Rationals, indetnames);;
indets:= IndeterminatesOfPolynomialRing(R);

t:= indets[1];;
x:= indets[2];;
y:= indets[3];;
z:= indets[4];;
```

Here is a 4x4 matrix over R in one indeterminate

```
o:= One( R );;
mat:= List([1..4], i -> [2^(i-1), 5^(i-1), 17^(i-1), x^(i-1)])*o;;
Display(mat);
Factors(Determinant(mat));
```

GAP will factor polynomials in one indeterminate but not polynomials with more than one indeterminate

3 magma

`magma` is also a fully developed tool with incredible capabilities. Unfortunately, it is not free, but it is available for reasonable cost; see `//magma.maths.usyd.edu.au/magma`.

In this introduction we summarize only the points which are most pertinent to computations in error correction coding, leaving untouched most of the vast capability of this tool.

Syntactically, all commands entered in `magma` must end with a semicolon `;`. `magma` will print the result if no computations are done. This is an easy way to print the result. Assignments are done using the `:=` operation. Scroll-back and editing commands are available using arrow keys or the emacs-like bindings (C-p for previous line, C-f for forward character, etc.). A help system is accessed using `?`.

This tutorial barely scratches the surface of the capabilities. Reference to the manual or online help us strongly suggested. Parts of this tutorial material is drawn from material posted at the website above.

An important aspect about `magma` is that every object within it must belong to some "space." Numbers are rationals, reals, integers, etc. Polynomials have coefficients from some specified field or ring. A vector subspace resides in some specific vector space. Once the "home" of the objects is established, the rest of the computations usually follows quite readily.

In some cases, it is necessary to coerce an object to be in some particular space. The coercion operator is `!`. For example, a sequence of numbers `[1,1,1,1]` has no particular "home." However, it can be interpreted as an element of a four-dimensional vector space with rational coefficients as follows:

```
V := VectorSpace(RationalField(),4);   // set up the vector space
v := V![1,1,1,1];                      // coerce to the right place
```

You can determine what space an object lives in using the `Parent` command.

```
Parent([1,1,1,1]);
Parent(Vector([1,1,1,1]));
Parent(V![1,1,1,1]);
```

3.1 Finite Fields and Polynomial Rings over Finite Fields

To create a finite field in `magma` use

```
F := FiniteField(5);
```

or

```
F := GF(5);
```

Polynomial computations are done in a specified ring, and you name the indeterminate variable when you create the ring. To create the ring $GF(5)[x]$ use

```
R<x> := PolynomialRing(F);
```

Then polynomials can be entered just as one might expect:

```
p := 3*x^2 + 4*x;
p;
```

In this case, the display format is pretty much like the input format. (This makes `magma` output much easier to read, in this author's opinion. It also makes input easier, because `magma` is able to figure out which field the coefficients belong to.)

Computations in $GF(2)$ are especially easy.

```
F := FiniteField(2);
R<x> := PolynomialRing(F);

(x^2 + x+1)*(x^2 + x^5 + x);
(x^2 + x+1) + (x^2 + x^5 + x);
(x^2 + x+1) - (x^2 + x^5 + x);
```

Now consider working over $GF(2^4)$:

```
F<a> := FiniteField(2^4);
```

In this case, the identifier `a` is returned as a primitive element in the field, the root of the primitive polynomial used to build the extension. In other words, it is α.

To determine the defining polynomial for an extension field:

```
p := DefiningPolynomial(F);
```

To get the generator,

```
a := Generator(F);
```

To evaluate a polynomial at a value

```
Evaluate(p,a);
```

To build an extension field using your own specified polynomial, use the following:

```
F2 := GF(2);    // create the field for the coefficients
R2<x> := PolynomialRing(F2);  // build GF(2)[x]
F := ext<GF(2) | x^4+x^3+1>;  // the polynomial is in in GF(2)[x]
```

Basic polynomial operations are straightforward:

```
(a^4*x^3 + a^2*x^2 + a^5*x + 1) * (a^8*x^2 + a^7*x + 1);
(a^4*x^3 + a^2*x^2 + a^5*x + 1) - (a^8*x^2 + a^7*x + 1);
```

Computing quotients (division without remainder) and remainders uses the following commands

```
(a^4*x^3 + a^2*x^2 + a^5*x + 1) div (a^8*x^2 + a^7*x + 1);
(a^4*x^3 + a^2*x^2 + a^5*x + 1) mod (a^8*x^2 + a^7*x + 1);
```

The GCD is accessed using `GCD`. The coefficients in the GCD are found using `XGCD` (extended GCD).

Factors of polynomials can be found using `Factorization`:

```
Factorization(p);
```

3.2 Vectors and Matrices in `magma`

We can define a vector as in the following examples:

```
v1 := Vector([1,3/2,-4/3,22/7]);
v2 := Vector([1,3,2,4]);
v3 := Vector([1/2, 1/4, 1/3, 3/4]);
```

(If you just use something like the following

```
v := [1,2,3,4];
```

you create a sequence; this is not a vector, but can be coerced into one.

To create the vector space defined by these vectors, first create the four-dimensional vector space in which they live:

```
V4 := VectorSpace(RationalField(),4);
```

then create the subspace spanned by `v1, v2, v3`:

```
V := sub<V4 | v1,v2,v3>;
```

To see if [1,1,1,1] is in this vector subspace, we must first coerce it to be a vector in the overriding vector space. Then we can use the `in` operator to check set membership

```
V4![1,1,1,1] in V;   // this is false: not in vector space V
V4![1,1,1,1] in V4;  // true: but it is in this space
```

Scalar multiplication is easy

```
v1*45;
```

as is the inner product

```
InnerProduct(V4!v1,V4!v2);
```

Vectors over a finite field. Let us construct the 3-dimensional vector space over $GF(3)$:

```
V := VectorSpace(GF(3),3);
```

and see if $[1, 1, 1]$ is in it:

```
V![1,1,1] in V;   // the answer is true
```

Bases Set up a vector space spanned by 4 vectors.

```
V4 := VectorSpace(RationalField(),4);  // overriding vector space
v1 := V4![2,2,1,3];
v2 := V4![7,5,5,5];
v3 := V4![3,2,2,1];
v4 := V4![2,1,2,1];
V := sub<V4 | v1,v2,v3,v4>;
```

A basis can be found with

```
Basis(V);
```

and the dimension can be found with

```
Dimension(V);
```

Given a vector $v \in V$ we can find the coefficients to represent it using the basis vectors of V as

```
Coordinates(V,v);
```

For example, for [2,1,2,1]

```
Coordinates(V,V4![2,1,2,1]);
```

If the vector v is not in the specified vector space, we get a runtime error

```
Coordinates(V,V4![1,1,1,1]);
```

Defining matrices We can easily make a matrix with integer elements:

```
M := Matrix([[1/2,2,3],[2,3,-1],[1,-2,5]]);
```

We can print and assign elements to an existing matrix using C-like notation

```
M[2][3] := 5;
M;
```

To build a matrix over $GF(5)$,

```
M := Matrix(GF(5),[[1/2,2,3],[2,3,-1],[1,-2,5]]);
Parent(M);
```

The identity matrix can be constructed (but you must indicate what field the coefficients belong to. For example

```
I := IdentityMatrix(RationalField(),3);
I := IdentityMatrix(GF(2^4),4);
```

A diagonal matrix is also easily constructed

```
D := DiagonalMatrix([a^2, a^3, a^4]);
```

Matrix algebra Addition, multiplication, etc., is straightforward. For example:

```
M := MatrixRing(GF(5),3);  // create the space of 3x3 matrices
m:= [];
m[1]:= M![1,2,3, 2,3,-1, 1,-2,5];
m[2]:= M![-1,4,-3, 1,2,-1, -1,-2,3];
m[3]:= m[1] + m[2];
m[4]:= m[1]*m[2];
m[5]:= 3*m[1] - 7*m[2];
m[6]:= m[1]^3;
m[7]:= m[2]^(-1);
m[8]:= m[6]*(4*m[5] + 6*m[2]*m[4])^(-1);

for i in [1..8] do
  print(m[i]);
  print("\n");
end for;

To find the transpose of a matrix use {\tt Transpose}
\begin{verbatim}
m:= M![1,2,3, 2,3,-1, 1,-2,5];
mdash:=Transpose(m);
```

Some determinants We use the **Determinant** function in the obvious way

```
mnew:= M![1,2,3, 2,3,-1, 1,-2,5];
Determinant( m1 );
```

If we look at matrices over GF(2) then their determinants will be either 0 or 1. What proportion would one expect for each with 2 2 matrices? Let us examine the situation.

```
countOne:= 0;;
countZero:= 0;;
M2 := MatrixRing(GF(2),2);
for i in [1..1000] do
  mat:= Random(M2);
   if Determinant(mat) eq 0 then
     countOne:= countOne + 1;
   else
     countZero:= countZero + 1;
   end if;
end for;
print countOne, " ", countZero, "\n";
```

Eigenvalues and eigenvectors are obtained with the commands `Eigenvalues` and `Eigenspace`, respectively.

```
Eigenvectors(m[1]);
Eigenspace(m[1],2);
```

Systems of linear equations To solve $XA = B$, where B is a row vector, we can use the `Solution` command.

```
Mr := MatrixAlgebra(RationalField(),3);
    // set up space for operations
A := Mr![1,2,1,1,-1,2,1,2,1];
B := Vector(RationalField(),[1,1,4/3]);
X := Solution(A,B);
X*A;  // check the solution
```

To get the nullspace of A (to obtain other possible solution) use two return arguments, as in

```
X,N := Solution(A,B);
```

Matrices over Polynomial Rings

```
R<x> := PolynomialRing(RationalField());
M := MatrixAlgebra(R,2);
mat := M![1,x,2+x,x^2];
```

Then the usual operations work:

```
mat^2;
mat + mat;
Determinant(mat^2);
```

However, inverses will not work, since they may lead to *ratios* of polynomials, not just polynomials. We have to tell `magma` to deal with this larger object.

```
R<x> := PolynomialRing(RationalField()); // set up polynomial ring
RF := RationalFunctionField(R); // set up rational function field
M := MatrixAlgebra(RF,2);
mat := M![1,x,2+x,x^2];
```

Now we can compute the inverse:

```
mat^(-1);
```

and `magma`.