An Introduction to

Magma

for Coding Theory

Giuseppe Cotardo

February 25, 2020

This notes are a summary of the book:

An Introduction to Algebraic Programming with Magma

by John J. Cannon and Catherine Playoust

Full version of the book is available at

https://magma.maths.usyd.edu.au/magma/pdf/intro.pdf

Contents

Part I

Overview

Chapter 1

Getting Started with Magma

1.1 Input and Output

Commands in MAGMA are known as **statements**. Each complete statement **must** finish with a **semicolon** (;). Magma will not perform the statement on the current line until the return key has been pressed.
Whenever MAGMA is ready to receive a statement from the user, it dis- plays a **prompt symbol** on the left of the input line. It looks like this:

```
>
```

Statements must be typed following the prompt symbol.

> **Example:**
> In order to find the sum of 2 and 3, the user should type `2+3;` after the prompt symbol and press the return key. MAGMA will execute the statement and return `6` as output. The screen will look like this:
>
> ```
> > 2+3;
> 6
> ```
>
> If output differs to the above on the user's screen, then a typing error has been made. The user should correct the error and try again. **If there is no output, the most likely reason is that the semicolon at the end of the line has been forgotten**. In this case, the user should type a semicolon, and then press the return key.

1.2 Magma Help System

MAGMA is documented in several ways: *online help*, *signatures* of *intrinsics*, hypertext help, and books. This section explains how to use the online MAGMA system and to read the information contained in the signature of the intrinsics.

1.2.1 Intrinsics and Signatures

The functions and procedures which form part of the MAGMA system are known as **intrinsics**. There are two different types of intrinsics, *system intrinsics* and *user intrinsics*, but they are indistinguishable in their use. A system intrinsic is a *function* or a *procedure* programmed by

the MAGMA developers and their associates, whereas a *user intrinsic* is an informal addition to MAGMA , created by a user of the system.
The **signature** of an intrinsic is a short description of the intrinsic, in a fixed format. First comes the list of its arguments and their categories, then (if it is a function) the list of the categories of its return values, then details of any parameters it has, and finally a line or two of text describing what the intrinsic does. The way to obtain the signature of an intrinsic is to print the intrinsic.

Example:
The following line shows how to obtain all the information about the intrinsic `Factorial`.

```
> Factorial;
Intrinsic 'Factorial'

Signatures:

      (n::RngIntElt) -> RngIntElt

         The factorial n! for small non-negative n.
```

This signature states that Factorial has one input n, in the **RngIntElt** category (that is, an integer) and returns an integer which represent the factorial of n.

Some intrinsics can take several kinds of arguments, with respect to their categories or the number of arguments, so they have several signatures.

Example:
For example typing

```
> Rank;
```

we read, among all the different signatures,

```
    (E::CrvEll) -> RngIntElt, BoolElt
    [
        Bound,
        Effort
    ]

        The Mordell-Weil rank of the elliptic curve E
        (or possibly a lower bound if the returned
        boolean is false).

    (D::CosetGeom) -> RngIntElt

        The rank of the coset geometry D.

    (R::FldFunRat) -> RngIntElt
    (R::GenMPolB) -> RngIntElt
    (R::GenMPolRes) -> RngIntElt
```

```
        The number of indeterminates of R over its
        coefficient ring.

    (W::GrpMat) -> RngIntElt
    (W::GrpPermCox) -> RngIntElt

        The rank of W.

    (H::HomModAbVar) -> RngIntElt

        The rank of H as a Z-module or Q-vector space.

    (phi::MapModAbVar) -> RngIntElt

        The dimension of the kernel of phi.

    (A::MtrxDiag) -> ModTupRng

        Return the rank of A.
```

These signatures show that the intrinsic `Rank` returns an integer, but that its argument can be: an element of a matrix algebra; an element of a homomorphism module; a tuple module; a homomorphism module; a univariate or multivariate polynomial ring or a quotient of such a ring; a lattice; or an elliptic curve.

1.2.2 Online Help

The online help system includes specially-written material, suitable for near-beginners, and complete access to the descriptions in the *Handbook*. This documentation can be consulted by visiting the following website:

`https://magma.maths.usyd.edu.au/magma/handbook/`

Alternatively, the user can call-up the handbook entry for an intrinsic by typing `?` followed by the name, after the prompt symbol, and pressing the return key. There is no need of the semicolon at the end of this statement.

Example:
To read the handbook entry about the intrinsic `Factorial`, the user should type:

```
> ?Factorial
```

1.3 Quitting Magma

The command for finishing a MAGMA session is

```
> quit;
```

The semicolon, followed by the return key, is compulsory. The MAGMA run will then be terminated. An alternative method of quitting is simply to close the terminal.

Chapter 2

The Language

This chapter describes some elementary features of MAGMA. It explains how to produce output by evaluating expressions, how to assign a value to an identifier, how to perform function calls, and how to handle common sub- expressions. In addition, it gives an overview of the operations on integers, rationals, reals, complex numbers, and Boolean values, and it shows how to include comments, recall previously printed values, and load input from a file.

2.1 Arithmetics

The **addition** and **subtraction** operators in MAGMA are represented by the `+` and `-` symbols, as would be expected. As is the case in algebra, these operators are left associative so that an expression consisting entirely of additions and/or subtractions is evaluated in order from left to right. Parentheses may be used to force a particular order of evaluation. For example:

```
> 2 - 3 + 4;
3
> 2 - ( 3 + 4 );
-5
```

Notice that the calculations above can be written as:

```
> 2 - 3 + 4 , 2 - ( 3 + 4 );
3 5
```

Spaces around operators and commas are optional.
The other standard arithmetic operations are present, but their notation may be unfamiliar.

- The **multiplication** operator is an asterisk `*`.
- The **exponentiation** operator is a caret `^`.
- There are three different operators for **division**:
  - the slash `/` operator, which performs exact division;
  - the `div` operator, which gives the quotient of the division;
  - the `mod` operator, which gives the remainder of the division.

Notice that the operators `div` and `mod` are defined for pairs of elements of Euclidean Rings. The slash operator is defined for pairs of elements of any ring provided that the divisor is a unit element. As a special case, exact division when applied to the integers a and b constructs the element $\frac{a}{b}$ of the rational field.

Example:
This example illustrates the division of 15 by 6 using the different operators in the integer ring.

```
> 15/6;
5/2
> 15 div 6;
2
> 15 mod 6;
3
```

Example:
This example illustrates the various division operators in the univariate polynomial ring over the integer ring.

```
> P<x> := PolynomialRing(IntegerRing());
>(x^5 + 4) div (x^2 - 3*x + 1);
x^3 + 3*x^2 + 8*x + 21
> (x^5 + 4) mod (x^2 - 3*x + 1);
55*x - 17
```

Using these operators, one can construct arbitrarily complicated arithmetic expressions. MAGMA's operator precedence for the order of evaluation corresponds to that used in arithmetic and algebra, but parentheses may be used to override this or to clarify the expression for the human reader.

2.2 Function Calls

An (intrinsic) function is evaluated for specific values of its arguments by typing its name followed by the list of arguments, separated by commas and enclosed within parentheses. If the intrinsic does not have any arguments, the parentheses must still appear.

Example:
To compute 5!, the function `Factorial` has to eveluated to the integer 5:

```
> Factorial(5);
120
```

In the following lines, the binomial coefficient 5 choose 2 is printed.

```
> Binomial(5,2);
10
```

The process of applying a function to a particular set of values is often referred to as *invoking* or *calling* the function.

2.3 Printing Text

None of the MAGMA outputs above included any explanation of what the printed values meant. It is often desirable to print some explanatory text together with a value. The text must be enclosed between double quotation marks ", (note this is a single character on the keyboard).

Example:

```
> "Factorial of five is", Factorial(5);
Factorial of five is 120
```

The object formed by enclosing characters in double quotation marks is called a **string**.

2.4 Identifier

2.4.1 Labelling

It could be useful for the user to store a piece of information in a 'box' in the computer's memory in order to reuse it later. For example, suppose one is given an integer 2104160246 which has to be used multiple times in the computations. In this case it is better for the user to store this value as typing mistakes can easily occur. It would much easier to type

```
> n*4, n*63, n*94;
```

rater than

```
> 2104160246*4, 2104160246*63, 2104160263*94;
```

provided that MAGMA can be told that m stands for the multiplier. The way to accomplish this is to type

```
> n:=2104160246;
```

The technical term for a label used to identify a piece of information stored in a computer is **identifier** or **variable**, and the process of giving the identifier a value is called assignment.
Once the identifier n has been assigned the value 2104160246, MAGMA will substitute this value wherever n appears in an expression. If n is later assigned a different value, this new value will be substituted in subsequent expressions involving n.

Example:

```
> n;
2104160246
> n * 4, n * 63, n * 94;
8416640984 132562095498 197791064722
```

An identifier may have as its value, any object definable in MAGMA : an integer, a group element, a set, a rational number, a string or a vector space. The main role of identifiers, other than for purposes of abbreviation, is that they allow the construction of more general program statements than those that can be formed using constants alone. Because of their utility, it is usual for many identifiers to be used in a large MAGMA program.

Identifier names are not restricted to a single letter, such as m. In fact, it is a good idea to choose longer and meaningful names. An identifier may be constructed from letters, digits or the underscore character _, provided that it does not commence with a digit. Thus, $side3$ and $angleA$ are examples of legal identifier names, while $3n$ is not. There are four traps to avoid in choosing identifiers:

- Magma distinguishes between upper and lower case letters. For example, rc, RC, rC and Rc are distinct identifiers.
- Although it is possible assign a value to an identifier that is the name of an intrinsic function (e.g., `Factorial`), this is very unwise since the intrinsic will no longer be available. It may be retrieved by applying the delete-statement to the intrinsic identifier which has been redefined.
- The identifier name `_` has a special use as the throwaway identifier, so it may not be used as an ordinary identifier name.
- A MAGMA reserved word such as `print` or `mod` may not be used as an identifier. All the reserved words are listed below.

`_`	`do`	`hom`	`notin`	`save`
`adj`	`elif`	`if`	`notsubset`	`sdiff`
`and`	`else`	`import`	`or`	`select`
`assert`	`end`	`in`	`print`	`subset`
`assigned`	`eq`	`instrinsic`	`printf`	`then`
`break`	`error`	`is`	`procedure`	`time`
`by`	`exists`	`join`	`quit`	`to`
`case`	`exit`	`le`	`random`	`true`
`cat`	`false`	`load`	`read`	`until`
`clear`	`for`	`local`	`readi`	`when`
`cmpeq`	`forall`	`lt`	`repeat`	`where`
`continue`	`forward`	`meet`	`require`	`while`
`default`	`freeze`	`mod`	`requirege`	`xor`
`delete`	`function`	`ne`	`reuquirerange`	
`diff`	`ge`	`not`	`restore`	
`div`	`gt`	`notadj`	`return`	

2.5 Show All Identifiers and Their Values

The intrinsic procedure **ShowIdentifiers()** prints a list of all the identifiers that are currently assigned. A similar procedure, **ShowValues()**, prints all these identifiers together with their values.

Example:

```
> Z := IntegerRing();
> d := 4;
> r := 5/3;
> ShowIdentifiers();
```

```
Z   d   r
> ShowValues();
Z:
    Integer Ring

d:
    4

r:
    5/3
```

2.6 Deletion and Unassigned Identifiers

If a Magma session has involved the creation of very large objects, or large numbers of smaller objects, process space (i.e., workspace) may be exhausted. In order to free memory one can remove unwanted data using the following statement:

`delete` identifier, identifier, ... , identifier;

This statement deletes the current values of all the listed identifiers, so that they revert to the status of identifiers that have not been assigned. The former values of those identifiers will be lost irrevocably from MAGMA's memory. The unary operator `assigned` allows the user to test whether an identifier is currently assigned a value. It returns true or false.

Example:

```
> a := 4;
> assigned a;
true
> delete a;
> assigned a;
false
```

2.7 Assignment

2.7.1 Simple Assignment

Various examples of assigning a value to an identifier have appeared above. The general form is:

identifier := espression;

where the expression involves identifiers and/or constants. When MAGMA encounters this statement, it evaluates the expression on the right hand side, stores the resulting value, and associates the identifier with that value. If an error occurs during the evaluation of the expression, the identifier will not be assigned a value.

Example:
Suppose hat the dimensions of a solid box are $L = 14$, $B = 12$ and $H = 6$, and the volume

of the box has to be stored in V . Firstly, the identifiers L, B and H should be assigned:

```
> L := 14;
> B := 12;
> H := 6;
```

Now V may be calculated in terms of these identifiers:

```
> V := L * B * H;
```

Then V may be accessed on demand.

```
> V;
1008
> "Volume is", V, "cubic units.";
Volume is 1008 cubic units.
```

and it can also be used to calculate other quantities such as the mass of the box, given that the box's density is 100 units:

```
> "Mass is", V * 100, "units.";
Mass is 100800 units.
```

It is important to realize that only a value is stored in an identifier, not the expression used to calculate the value. For example, V will not change if L is now changed:

```
> L := 57;
> V;
1008
```

The expression on the right side of an assignment statement sometimes involves the identifier on the left side. A common example of this is

```
n := n + 1;
```

The statement simply means that Magma should retrieve the current value of n, add 1 to it, and store the result as the new value of n. It is not related to the inconsistent equation $n = n + 1$. This is the reason why the Magma assignment symbol is := rather than =. The user should regard the symbol := as meaning 'becomes' or 'has assigned to it', rather than 'is equal to' in the mathematical sense.
A final caution: if an identifier is assigned a value and then assigned another value, the old value is lost and cannot be recovered. The classic case occurs in trying to interchange the values of two identifiers, a and b. It seems at first that the following lines will suffice, but they do not:

```
> a := b;
> b := a;
```

The correct solution is to use a temporary identifier (called *temp*, say), as follows:

```
> temp := a;
> a := b;
> b := temp;
```

In this way a can acquire b's former value and b can acquire a's former value, without either of them being lost in the process.

2.7.2 Mutation Assignment

There is an alternative form of assignment statement available for making a simple change to an existing identifier that involves its former value. Suppose that the following assignment has been made:

```
> d := 7;
```

and then d has to be multiplied by 53. Instead of typing

```
> d := d * 53;
```

the user can type

```
> d *:= 53;
```

This is known as a **mutation assignment** because it mutates, or changes, the old value of the identifier. Its general form is

identifier ∘ := expression;

where ∘ is the operator, and it has the same end result as

identifier := identifier ∘ expression;

Apart from being more compact than the standard assignment, a mutation assignment is often more efficient since MAGMA can operate directly on the identifier's value without first having to make a copy of it.

2.7.3 Multi-Valued Expressions and Multiple Assignment

Certain classes of expression in MAGMA may return multiple values. These expression are called **multi-valued expressions**. The general form of the statement which allows to access he values returned by a multi-valued assignment is:

identifier,..., identifier := multiple-value expression;

Example:
The function `Quotrem`(a, b) returns the values of both a `div` b and a `mod` b

```
> q, r := Quotrem(26, 4);
> q, r;
6 2
> 4*q + r;
26
```

If such an expression forms part of a larger expression, then only the principal value (the first return value) will be used.

```
> 5 * Quotrem(100, 13);
35
```

The number m of identifiers on the left of the multiple assignment statement must be less than or equal to the total number n of return values. If m is strictly less than n, then only the first m return values of the expression will be returned and assigned. The special character _ (known as the **throwaway identifier**) may be used in the identifier list to discard the corresponding return value.

```
> q := Quotrem(100, 13);
> q; 7
> _, r := Quotrem(100, 13);
> r;
9
```

It should be noted that in the case of functions for which significant additional work has to be done to compute non-principal return values, this extra computation may be avoided by requesting only the first return value (by assigning only to a single identifier). So to avoid unnecessary work, values other than the first should not be requested unless they are required. An example is the `SmithForm`(X) function, whose principal return value is the *Smith Normal Form* S of the matrix X. Its second and third values are *unimodular matrices* P and Q such that $PXQ = S$, but since these matrices take significantly longer to compute than S, they are only calculated if the user requests them specifically. The example below demonstrates the time savings for a random 300×300 integer matrix X, each of whose coefficients is -1, 0 or 1. Note that the `time` command, placed at the beginning of a statement, has the effect of printing the execution time (in seconds) for that statement.

```
> n := 300;
> M := MatrixRing(IntegerRing(), n);
> M;
Full Matrix Algebra of degree 70 over Integer Ring
> X := M ! [Random(-1, 1): i in [1 .. n^2]];
> time S := SmithForm(X);
Time: 1.130
> time S, P, Q := SmithForm(X);
Time: 6.420
```

2.7.4 The where-construction

Often a MAGMA expression will include the same subexpression more than once. It is faster for both the user and the system to assign the value of that expression to a temporary identifier. This may be accomplished with the **where-construction**, which has the form expression

`where` identifier `is` expression

This entire construct is understood as constituting a single expression. Its value is found by evaluating the first expression, with the given identifier taking as its value, the value of the second expression.
The scope of the identifier in the where-construction is limited to that construction, so it does not affect an identifier of the same name outside it.

Example:

```
> P<x> := PolynomialRing(IntegerRing());
> d := 1000;
> f := d*(d + 1)*(d^2 + 5) where d is x^5 - 4*x^3 + 17;
> f;
x^20 - 16*x^18 + 96*x^16 + 69*x^15 - 256*x^14 - 828*x^13 + 256*x^12
    + 3312*x^11 + 1790*x^10 - 4416*x^9 - 14320*x^8 + 28640*x^6
    + 20694*x^5 - 82776*x^3 + 89964
> d;
1000
```

Notice that d has kept its old value. If d had not been assigned, then after the where-construction it would have remained unassigned.

Example:
Several where-constructions may be used in succession. If an expression involves successive where-clauses, they associate to the left.

```
> x := 1; y := 2;
> x + y where x is 5 where y is 7;
12
> (x + y where x is 5) where y is 7;
12
```

In the case of a comma-separated expression list, the effect of a where-clause will extend left across the commas, but not right.

Example:
Notice that the first two expressions in the second line are evaluated using the values for x and y given in the where-constructions, but the other expression is evaluated using the external values $x = 1$ and $y = 2$:

```
> x := 1; y := 2;
> x-y, x+y where x is 5 where y is 7, x*y;
-2 12 2
```

2.8 Integers, Rationals, Reals, and Complex Numbers

Arithmetic with integers has already been demonstrated. Rational numbers in MAGMA are expressed in fractional form, and real numbers in decimal form:

```
> halfRat := 1/2;
> halfReal := 0.5;
```

A rational number such as $20 + \frac{4}{7}$ may be entered as either $20 + 4/7$ or $144/7$. A real number having very large or very small magnitude may be entered in scientific notation, using e to denote the base 10. For example, the input syntax for 5.2×10^{-16} is:

```
> smallnum := 5.2e-16;
```

A complex number is best created as an expression involving two real numbers and $i = \sqrt{-1}$, where i is created using the `ComplexField`() function:

```
> C<i> := ComplexField();
> z := 2.6 + 3.7*i;
```

Rational arithmetic in MAGMA is performed to arbitrary precision, since it is based internally on integer arithmetic. Real/complex arithmetic is performed to free precision or to a fixed precision, depending on the setting of the default real field.

MAGMA can usually cope easily with an expression involving both an integer and a rational, real or complex number, by coercing the integer into the appropriate field. However, for some applications MAGMA must be told explicitly that an integer is to be considered as a real, rational or complex number. The easiest way of doing this to write it as such, as shown below:

```
> rat37 := 37/1;
> real37 := 37.0;
> cmplx37 := 37.0 + 0*i;
```

These numbers will be deemed by MAGMA to be equal to the integer 37, but the structure to which they belong will be the rational, real or complex field rather than the integer ring. The function `Parent`() will return the structure that the object belongs to:

```
> Parent(rat37);
Rational Field
> Parent(real37);
Real field of precision 30
> Parent(cmplx37);
Complex field of precision 30
```

An alternative way to construct *rat*37, *real*37 and *cmplx*37 is to coerce them explicitly into the required structure, using the `!` operator:

```
> rat37 := RationalField() ! 37;
> real37 := RealField() ! 37;
> cmplx37 := C ! 37;
```

Integers, rationals, reals and complex numbers share most of the common arithmetic operators, but, in addition, each has its own special functions.

Example:

```
> Numerator(r), Denominator(r) where r is 6/4;
32
> Sin(4.6);
-0.99369100363346445613810465991370046463O
> Round(x), Truncate(x) where x is 3.7;
4 3
> Sqrt(-3); // square root
1.732050807568877293527446341495*i
```

```
> ComplexToPolar(3 + 4*i); // modulus and argument
5.0000000000000000000000000000
0.92729521800161223242851246291
```

2.9 Booleans

One of the most basic structures available in Magma is the **Boolean structure**, so named after the logician George Boole. The Boolean structure contains two values: **true** and **false**. Boolean values are used to describe a two-state system, such as yes/no, on/off, or true/false. There are several operators that return a Boolean value. Some examples of **relation operators** are listed in the following table[1].

Operator	Maths	Usage	Meaning
`eq`	$=$	x `eq` y	**true** iff x is equal to y
`ne`	$\neq$	x `ne` y	**true** iff x is not equal to y
`lt`	$<$	x `lt` y	**true** iff x is strictly less than y
`le`	$\leq$	x `le` y	**true** iff x is less than or equal to y
`gt`	$>$	x `gt` y	**true** iff x is strictly greater than y
`ge`	$\geq$	x `ge` y	**true** iff x is greater than or equal to y

These relational operators test if two quantities are related in a certain way and they then return an answer of **true** or **false**.

Example:

```
> 3+4 eq 7;
true
> 100 eq 10^3;
false
> 2/3 lt 3/4;
true
```

The following is a list of the **Boolean operators**. They operate on Boolean values and return Boolean results.

Operator	Maths	Usage	Meaning
`not`	$\neg$	`not` x	**true** iff x is **false**
`and`	$\wedge$	x `and` y	**true** iff both x and y are **true**
`or`	$\vee$	x `or` y	**true** iff at least one of x and y is **true**
`xor`	$\veebar$	x `or` y	**true** iff exactly one of x and y is **true**

The meanings of the Boolean operators correspond fairly closely to the ordinary meanings of the corresponding words, except that that `or` is inclusive (true if and only if one or both of a and b are true), whereas `xor` is exclusive (true if one of a and b is true but not if both are true). In other words, `or` is in accordance with the use of 'or' in 'This seat is reserved for aged or disabled persons' and `xor` corresponds to 'or' in 'Would you like coffee or tea?'.

[1] Note that 'iff' is an abbreviation for 'if and only if'.

Example:

```
> (3+4 gt -5) and (2 le 3);
true
> (3+4 gt -5) or (2 le 3);
true
> (3+4 gt -5) xor (2 le 3);
false
```

There are many functions in Magma that test whether their argument(s) satisfies some property, and return a Boolean value as the result. Boolean functions typically have names beginning with **Is** (or **Has** or **Are**) followed immediately by the name of the property to be tested.

Example:
The function `IsIndependent`(S) returns **true** iff S is a set of linearly independent elements of the vector space V:

```
> V:=VectorSpace(GF(2),3);
> u:=V![1,0,0];
> w:=V![0,1,0];
> S:={u,w};
> IsIndependent(S);
true
```

Some Boolean functions have additional return values that provide further information.

Example:
The function `IsSquare`(a) returns **true** and the positive square root of a, if the integer a is a perfect square, and returns **false** alone otherwise:

```
> sq, sqroot := IsSquare(144);
> sq;
true
> sqroot;
12
> sq, sqroot := IsSquare(143);
> sq;
false
> assigned sqroot;
false
```

2.10 Comments

It is standard practice to include comments in a computer program to describe its purpose and its internal logic. Such remarks are ignored by the computer.
A comment is placed between the `/*` and `*/` symbols. MAGMA will ignore anything between these symbols, and the symbols themselves. Note that comments may not be nested. Alternatively, any text following the symbols `//` on a line will be ignored (treated as a comment).

Example:

```
> /*
> * Find the volume of a box,
> * given its length, width, and height.
> */
> // assign the dimensions
> L := 14; // length
> W := 12; // width
> H := 6; // height
> // compute the volume
> V := L * W * H;
```

2.11 Recalling Previously Printed Values

It frequently occurs that a user requires a value printed earlier, which has not been assigned to an identifier. Since it would be inefficient to recompute the value, MAGMA provides a facility for recalling values that have been obtained by evaluating expression.

The list of values displayed by the most recent statement corresponds to `$1`, the list of values in the next most recent statement corresponds to `$2`, and so on. The number after the dollar sign must be a literal integer, not a general expression evaluating to an integer.

Example:

```
> z := 15;
> 3 * z;
45
> y := 13;
> $1;
45
> z * y + 1;
196
> $1 - $2; // i.e., 196 - 45
151
```

If several values were previously printed in a single statement, then the dollar construct will return all of them exactly as if it were a multi-valued expression, so the values may be assigned using the multiple assignment statement.

Example:

```
> 12 * 9, 4 * 8;
108 32
> a, b:= $1;
> a;
108
```

```
> b;
32
>
> Quotrem(5000, 183);
27 59
> c, d := $1;
> c;
27
> d;
59
```

Chapter 3

Aggregate Structures

Aggregate structures are used in MAGMA in order to gather objects together into one object so that they may be operated on as a body. There are many categories of aggregate structures, the principal ones being enumerated sets and enumerated sequences. This chapter describes the features of each kind of aggregate structure, and explains how to construct and operate upon them.

The greatest emphasis is given to the homogeneous iterable aggregate categories, that is, those categories such that the elements of each aggregate all have the same parent magma, and may be produced on demand by the system. Following these, formal sets (homogeneous non-iterable unordered aggregates without repeated elements) and lists (non-homogeneous iterable ordered aggregates) are discussed.

3.1 Properties of the Aggregate Categories

The following table lists some of the categories of aggregates.

Bracketing	Category	Meaning
`{...}`	`SetEnum`	Enumerate set
`{*... *}`	`SetMulti`	Multiset
`{@... @}`	`SetIndx`	Indexed set
`[...]`	`SeqEnum`	Enumerated sequence
`[* ... *]`	`List`	List

Notationally, the aggregate categories are distinguished by different kinds of bracketing symbols surrounding the description of the aggregate's contents; the table includes these bracketing symbols for reference. Most of MAGMA's aggregate categories will be familiar from mathematical contexts. The exceptions are records and record formats, which are primarily programming constructions. For this reason, the discussion of records and record formats will be deferred until the end of this chapter.

Several attributes distinguish the other categories of aggregates. The chief features of an aggregate category C are whether, for every aggregate S in C, S is homogeneous, iterable, ordered, and/or capable of containing repeated elements. These features will now be briefly explained.

- A **homogeneous** aggregate is a collection of objects which all have the same parent magma. This common parent is called the **universe**.
- An **iterable** aggregate is a finite aggregate all of whose elements $S_1, \ldots, S_n$ MAGMA can produce successively on demand.
- An aggregate with **repeated elements** is one which does not satisfy the property that all its elements are distinct.
- An **ordered** aggregate is an aggregate S supplied with an index map from $\{1, \ldots, n\}$ to S, where n is the cardinality of S. In MAGMA , the i-th element of an ordered aggregate S is notated `S[i]`, and whenever iteration occurs over S it will be in the order `S[1],...,S[n]`, 'from left to right'.
- An **enumerated** set in MAGMA corresponds to the mathematical concept of a set as an unordered collection of distinct objects, except that it must be finite and homogeneous.

3.2 Construction of Iterable Homogeneous Aggregates

Enumerated sets, multisets, indexed sets and enumerated sequences (the it- erable homogeneous aggregate categories) share many features with respect to construction. There are two main ways to create an aggregate structure belonging to one of these categories: from a list of expressions whose values are the elements; or from an element-description, given in set-theoretic notation, which is then evaluated by the system into a collection of elements. For the sake of convenience and efficiency, there is a special construction method for enumerated sets and sequences consisting of an arithmetic progression of integers, and these structures are stored as a virtual collection of elements.
Let S be an enumerated set, multiset, indexed set or enumerated sequence. If $e_1, \ldots, e_k$ are expressions evaluating to the elements of S, then S may be constructed by means of a comma-separated list of the e_i, enclosed in the appropriate bracketing symbol for the category of S.

Example:

```
> odds := {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
> odds;
{ 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 }
>
> q1 := [4, 8/3, 7/10, 33, 2, 1/2];
 > q1;
 [ 4, 8/3, 7/10, 33, 2, 1/2 ]
> ms := {* 2, 2, 3, 6, 1, 3, 3, 5, 3 *};
> ms;
{* 1, 2^^2, 3^^4, 5, 6 *}
```

As the printed version of the multiset ms suggests, an element of a multiset may be supplied together with its multiplicity, using the `^^` operator. In an element list for a multiset, `e^^m` has the same effect as m repetitions of e in the list. The integer m is called the multiplicity of e. Given a multiset S and a suitable object x, the function `Multiplicity`(S, x) returns the multiplicity of x in S, or 0 if x is not in S.

Example:

```
> ms2 := {* (a+4*c)^^4, b+c, 2*c, 9*a+b-5*c, 7*a^^2 *}; > print ms2;
{*
a + 4*c^^4, 2*c, 7*a^^2,
b + c,
9*a + b - 5*c *}
> print Multiplicity(ms2, a+4*c);
4
> print Multiplicity(ms2, 1000*b);
0
```

Both indexed sets and sequences have an ordering on their elements. If an indexed set or sequence S is created by means of an expression list, the resulting ordering corresponds to the ordering of the values in the list (after duplicates are removed, in the case of indexed sets). In particular, `S[i]` is the i-th element of S, counting from the left. The indexing commences at 1, not at 0 as in some programming languages.

Example:

```
> q1[5];
2
```

When an empty aggregate is used in computing, the implication is generally that it is a structure containing some particular type of object but that the number of objects in it happens to be zero. In MAGMA , an iterable homogeneous aggregate of this sort is defined by using bracketing symbols and declaring the universe, but not listing any objects.

Example:
An empty enumerated set of integers may be created as follows.

```
> emptyZ := { IntegerRing() | };
> emptyZ;
{}
> Parent(emptyZ);
Set of subsets of Integer Ring
```

Magma follows common usage and calls such structures empty. When a set or sequence becomes empty as the result of modifications, MAGMA keeps its universe defined and deems it to be empty in the MAGMA sense. The only elements that can be put into an empty set or sequence are those that can be made to fit the universe of that set or sequence. This prevents the user from including the wrong kind of element in the set; for example, putting a permutation group element into an empty set intended for vectors.

The term MAGMA uses for an iterable homogeneous aggregate that is not only empty but has no universe is null. It is created with bracketing symbols, without specifiying the universe or any elements. Any kind of element can be put into a null structure, but after that the structure will have a universe, like every set or sequence that contains elements.

The Boolean functions testing whether a set or sequence S is empty or null are `IsEmpty`(S) and `IsNull`(S) respectively. Note that a null set or sequence will return **true** when passed as an argument to `IsEmpty`, because it has no elements.

3.3 Operations on Iterable Homogeneous Aggregates

Several operations apply to more than one of the iterable homogeneous ag- gregate categories (enumerated sets, multisets, indexed sets, and enumerated sequences). These operations will be explained in this section.

3.3.1 Index Position, Cardinality and Length

It was mentioned above that if S is an indexed aggregate (indexed set or enumerated sequence), then `S[i]` returns the element of S at the ith index position. Conversely, `Position`(S, x) or `Index`(S, x) returns the index of the element x in S, or zero if x is not an element of S. If S is a sequence and x occurs more than once in S, the function returns the smallest such index, that is, the position of the leftmost occurrence of x in S.

Example:

```
> ii := {@ 8, 6, 6, 6, 6, 4, 6, 7, 2 @};
> ii;
{@ 8, 6, 4, 7, 2 @}
> ii[4];
7
> Position(ii, 7);
4
>
> qq := [ 8, 6, 6, 6, 6, 4, 6, 7, 2 ];
> qq;
[ 8, 6, 6, 6, 6, 4, 6, 7, 2 ]
> Position(qq, 7);
8
> Position(qq, 6);
2
```

The operator `#` is used throughout the MAGMA system to extract the cardinality of a magma S. If S is an object with n elements, where n is finite and can be calculated by MAGMA , then `#S` returns n. In the context of aggregate structures, the cardinality operator requires careful interpretation. For an enumerated set or indexed set S, in which all the elements are distinct, `#S` returns the number of these elements. For a multiset or an enumerated sequence S, in which there may be more than one occurrence of the same element in S, `#S` returns the total number of elements, counting multiplicities. Notice that from the point of view of the indexed aggregates, this definition implies that `#S` is the length or highest index of S; the same applies to `#S` where S is a **string**. Therefore `S[#S]` is the 'last' or rightmost element of S.

Example:

```
> m := {* wd[i]: i in [1..#wd] *} where wd is "MISSISSIPPI";
> m;
{* M, I^^4, P^^2, S^^4 *}
> #m;
11
```

3.3.2 Testing Equality, Membership, and Subsetness

The relational operators `eq` (=) and `ne` ($\neq$) may be applied to two aggregates R and S in the same category to test their equality/inequality. The universes of R and S must be compatible, that is, they must either be equal or have a common overstructure into which elements of each aggregate can be automatically coerced. In order for R eq S to return **true**, the elements of R and S must be equal, counting multiplicities where that is relevant.

Example:
The following two sets are regarded as equal by MAGMA , because their elements are exactly equal after the integers in the set on the right are coerced into the finite field.

```
> {FiniteField(7) | n: n in {2..5}} eq {2..5};
true
```

The expressions `x in S` and `x notin S` are used to test whether a given object x is an element of ($\in$) an aggregate S.

The expressions `R subset S` and `R notsubset S` compare two aggregates R and S to see whether R is a subset ($\subseteq$) of S, that is, whether every element of R is an element of S.

3.3.3 Including or Excluding an Element

The intrinsics `Include` and `Exclude` are used to place an element x in an aggregate S or take it from S. The parent of x must be compatible with the universe of S. There are two versions: the functions `Include`(S, x) and `Exclude`(S, x), which return a new aggregate but do not change the given aggregate S, and the procedures `Include`$(\sim S, x)$ and `Exclude`$(\sim S, x)$, which modify S itself, as a reference argument. If the user wishes to change the value of an existing aggregate, stored in an identifier, then the procedure version should be used; note that a procedure call is a statement by itself, and must be followed by a semicolon.

3.3.4 Maximum and Minimum Elements

The functions `Maximum`(S) and `Minimum`(S), which have abbreviations `Max`(S) and `Min`(S), may be applied to an aggregate S whose universe has a total ordering defined on its elements. These functions return the maximum or minimum element of S. If S is an enumerated sequence or an indexed set, the functions have a second return value, namely the index of the (leftmost) position where the maximal or minimal element occurs. A multiple assignment statement should be used to obtain both values.

3.4 Further Operations on Set Categories

The binary operations of union and intersection may be applied to two aggregates R and S in the same set category, provided that their universes are compatible. The aggregate returned will have the same category as the category of the arguments. The expression `R join S` returns the aggregate $R \cup S$, and `R meet S` returns $RcapS$. If R and S are indexed sets, no generalizations can be made about the indexing of the result. If R and S are multisets, `R join S` is constructed by adding multiplicities, and `R meet S` is constructed by taking the minimum of the multiplicities for each element.

```
> print { 3..9 by 2 } join { 1, 5, 11, 17 };
{ 1, 3, 5, 7, 9, 11, 17 }
> {* 1^^4, 3^^6, 8^^2 *} join {* 1, 2^^3, 8^^5 *};
{* 1^^5, 2^^3, 3^^6, 8^^7 *}
> {* 1^^4, 3^^6, 8^^2 *} meet {* 1, 2^^3, 8^^5 *};
{* 1, 8^^2 *}
```

There are two other binary set operators that are available only for enumerated sets and indexed sets, not for multisets. The expression `R diff S` returns the difference of R and S, that is, $\{x : x \in Randx \notin S\}$, and the expression `R sdiff S` returns the symmetric difference of R and S, that is, $\{x : x \in R \cup S\text{and}x \notin R \cap S\}$.

Example:

```
> { 3..9 by 2 } diff { 1, 5, 11, 17 };
{ 3, 7, 9 }
> { 3..9 by 2 } sdiff { 1, 5, 11, 17 };
{ 1, 3, 7, 9, 11, 17 }
```

For enumerated sets only, there are several functions provided that con- struct subsets, sub-multisets, permutations, and so on. Perhaps the most generally applicable of these functions is `Subsets`(S, k). Given an enumerated set S and an integer k, it returns the set of all k-subsets of S, that is, all subsets of S having cardinality k. If $k < 0$ or $k > \#S$, the value returned is an empty set.

Example:

```
  > Subsets({15, 8, 13, 42, 20}, 2);
{
    { 15, 20 },
    { 8, 13 },
    { 8, 42 },
    { 13, 15 },
    { 13, 20 },
    { 8, 15 },
    { 8, 20 },
    { 20, 42 },
    { 15, 42 },
```

```
    { 13, 42 }
}
```

3.5 Further Operations on Enumerated Sequences

This section discusses operations that apply to enumerated sequences only; see the following table for summaries. The operations in the table are functions or operators that return values; in order to mutate (change) the arguments, the user should type $\sim$ before the enumerate sequence (Q) in a manner corresponding to the non-mutation versions. For efficiency, the mutation operations should be employed where appropriate in preference to the functional versions.

MAGMA	Meaning
`Q cat T`	Sequence formed by chaining sequence T to end of sequence Q: $[q_1, ..., q_n, t_1, ..., t_m]$
`Position(Q, x)`, `Index(Q, x)`	Position of first occurrence of x in sequence Q, or zero if no term in Q is x
`Q[i]`	i-th term of sequence Q
`Q[I]`	$[q_{i_1}, \dots, q_{i_n}]$, where I is integer sequence $[i_1, \dots, i_n]$
`Q[i1,i2,...,ik]`	$Q[i_1][i_2] \dots [i_k]$ (multi-indexing)
`Explode(Q)`	The terms of Q, in index order
`Append(Q, x)`	Q with x placed on end
`Prune(Q)`	Q with final term removed
`Include(Q, x)`	Q with x placed on end, if x not in Q already
`Exclude(Q, x)`	Q with first occurrence of x in Q deleted, if x in Q
`Insert(Q, i, x)`	Q with x inserted at position i, and following elements moved along one place: $[q_1, \dots, q_{i-1}, x, q_i, \dots, q_n]$
`Remove(Q, i)`	Q with i-th term removed: $[q_1, \dots, q_{i-1}, q_{i+1}, \dots, q_n]$
`Insert(Q, k, m, T)`	Q with sequence T inserted, replacing terms q_k through to q_m inclusive: $[q_1, \dots, q_{k-1}, t_1, \dots, t_m, q_{m+1}, \dots, q_n]$
`Reverse(Q)`	Q with order of terms reversed
`Rotate(Q, p)`	Q with terms rotated cyclically p terms to the right, or $-p$ terms to the left if p negative
`Sort(Q)`	Q with terms sorted into increasing order

3.5.1 Changing a Term of a Sequence

Let Q be an enumerated sequence. It has already been seen that the expression `Q[i]` returns the i-th term of Q, and that `Position`(Q, x) or `Index`(Q, x) returns the index of the leftmost occurrence of x in Q (or 0 if x is not in Q). It is possible to change Q at the i-th term, by assigning a value to `Q[i]`.

Example:

```
> fr := [x/10: x in [0..10]];
> fr[6] := 99997;
> print fr;
```

```
[ 0, 1/10, 1/5, 3/10, 2/5, 99997, 3/5, 7/10, 4/5, 9/10, 1 ]
```

3.6 Lists

The purpose of the category `List` is to give the user a means to collect together several objects of assorted kinds. This would generally be done as a temporary measure during a larger task. Aggregates in this category are called **lists**, but this is a technical usage of the word, not to be confused with the informal use of 'list'.

Lists are ordered, iterable, and capable of containing repeated elements. However, their elements do not have to belong to the same magma, or even the same category. Therefore lists are like enumerated sequences, except without the restriction of homogeneity; the list bracket `[*` is intended to suggest this, by its similarity to the plain bracket `[` for enumerated sequences. However, enumerated sequences have more operations than lists and are much more efficient, so they should be preferred to lists whenever possible. The following table lists the operations on lists.

MAGMA	Meaning
`#L`	Length of list L (number of elements)
`IsEmpty(L)`	**true** if L is empty (has zero length)
`L[i]`	i-th term of list L ($1 \leq i \leq \#L$)
`L[i] := x`	Changes L by assigning x to the i-th term ($1 \leq i \leq \#L+1$)
`L[i1,i2,...,ik]`	`L[i1][i2]...[ik]` (multi-indexing)
`L cat K`	List formed by chaining list K to end of list L
`L cat:=K`	Changes L to (L `cat` K)
`Append(L, x)`	L with x placed on end
`Append(~L, x)`	Changes L to L with x placed on end
`Prune(L)`	L with final term removed
`Prune(~L)`	Changes L to L with final term removed

The only way to create a list is to 'list' its elements within the symbols `[*` and `*]`, since the more sophisticated constructors are not available.

Example:

```
> L := [* GF(8), "r", 1/3 *];
> L;
[* Finite field of size 2^3, r, 1/3 *]
> Category(L);
List
```

The operations on lists are an abridged version of the operations on enumerated sequences. The principal operations are finding the length of a list, accessing and changing the i-th term, concatenation, appending and pruning.

3.7 Transfer Functions Between Aggregates

It occasionally happens that a single aggregate category does not offer all the operations required for certain manipulations of a collection of objects. For such cases, MAGMA has transfer functions, which convert an aggregate from one category to another, preserving its elements. As far as possible, given the properties of the old and new categories, the order and multiplicity of the elements are retained. he transfer functions between aggregates are listed in the table below.

MAGMA	Meaning
`SetToSequence(S)`, `Setseq(S)`	Enumerated sequence corresponding to enumerated set S
`SequenceToSet(S)`, `Seqset(S)`	Enumerated set corresponding to enumerated sequence S
`SetToMultiset(S)`	Multiset corresponding to enumerated set S
`MultisetToSet(S)`	Enumerated set corresponding to multiset S
`SetToIndexedSet(S)`	Indexed set corresponding to enumerated set S
`IndexedSetToSet(S)`, `Isetset(S)`	Enumerated set corresponding to indexed set S
`IndexedSetToSequence(S)`, `Isetseq(S)`	Enumerated sequence corresponding to indexed set S
`SequenceToMultiset(S)`	Multiset corresponding to enumerated sequence S
`SequenceToList(S)`, `Seqlist(S)`	List corresponding to enumerated sequence S
`TupleToList(S)`, `Tuplist(S)`	List corresponding to tuple S

Notice that `Set` or `Sequence` within the name of a transfer function designate an enumerated set or enumerated sequence respectively. Several of the functions have abbreviated forms.

Example:
Suppose that q is an enumerated sequence of polynomials in x over $\mathbb{Z}$, and that the user wants to know how many times q contains the element $3x+5$. Although this operation could be done using sequence constructors, it may be more convenient to convert q to a multiset, using the function `SequenceToMultiset`, and then invoke the `Multiplicity` function.

```
> P<x> := PolynomialRing(IntegerRing());
> q := [ 6*x^2, 5*x+4, 3*x+5, 6*x^2, 6*x^2, 3*x+5, x^3-1,
> 3*x+5, 17*x^2-x+2, -2*x^3, 6, x-4 ];
> mq := SequenceToMultiset(q);
> mq;
{*
17*x^2 - x + 2, x^3 - 1, -2*x^3,
5*x + 4,
6,
x - 4,
3*x + 5^^3, 6*x^2^^3
*}
> Multiplicity(mq, 3*x+5);
3
```

> It may be concluded from the output that $3x + 5$ occurs three times in q.

Chapter 4

Conditional and Iterative Statements

4.1 Conditional Statements

One of the major purposes of Boolean values is to manage the flow of control in a program. Conditional statements and expressions use the results of Boolean tests in order to decide which statement to perform next, or which expression to evaluate. The main conditional statement in MAGMA is the **if**-statement. In this section the **case**-statement is also introduced.

4.1.1 The if-statement

Basic Form

MAGMA's **if**-statement causes MAGMA to perform different actions depending on whether a certain condition is **true** or **false**. The most basic form of the **if**-statement is:

```
if condition then
   statements
else
  statements
end if;
```

The statements between then and else are executed if the condition is **true**, and the statements between else and end if are executed if the condition is **false**. Note that, like all other statements, the **if**-statement ends with a semicolon. The newlines and indentation are optional, and are ignored by MAGMA , but they tend to make the code easier to read.

Example:
Suppose that a program is being written in which the absolute value of an identifier x has to be calculated and stored in y. If $x \geq 0$, y should equal x, but otherwise y should become the negation of x. The Boolean condition x `ge` 0 is appropriate for such a situation. To test an **if**-statement performing this action, x must be assigned a value (say -17) before the **if**-statement is executed:

```
> x := -17;
> if x ge 0 then
if> y := x;
if> else
```

```
if> y := -x;
if> end if;
> y;
17
```

The effect of this segment of code could also be achieved with MAGMA's intrinsic function `Abs`(r):

```
> y := Abs(x);
> y;
17
```

Notice that, as in the example above, while the **if**-statement is being typed, MAGMA provides a *special prompt* `if>` as a reminder that the statement has not yet closed.

Short Form

If a program requires an **if**-statement for which there is no action to be taken if the condition is false, a shorter form of the if-statement should be used. This shorter form simply omits the else clause:

```
if condition then
   statements
end if;
```

Example:
The absolute value code above could be rewritten as follows:

```
> x := -54;
> y := x;
> if x lt 0 then
if> y := -y;
if> end if;
> y;
54
```

Nested Form

Sometimes the else clause of an if-statement will itself be an **if**-statement. At the end of such an **if**-statement there would be matching `end if;` closures. This type of nested **if**-statement may be avoided by making use of `elif`, which is a shorter form of `else if` that also removes the need for an `end if;` to close off the inner **if**-statement:

```
if condition then
   statements
elif condition then
   statements
elif condition then
   statements
⋮
```

```
else
   statements
end if;
```

Example:
An **if**-statement with `elif` clauses may be used to convert from a percentage mark into a grade. The program assumes that mark has been assigned an integer in the range $0, \ldots, 100$:

```
> if mark ge 85 then
if> print "High Distinction";
if> elif mark ge 75 then
if> print "Distinction";
if> elif mark ge 65 then
if> print "Credit";
if> elif mark ge 50 then
if> print "Pass";
if> else
if> print "Fail";
if> end if;
```

4.1.2 A Long Example: Area of a Triangle

The next example of the **if**-statement is a procedure `TriangleArea`(a, b, c) that, given the side-lengths a, b, c of a triangle, calculates and prints the area of the triangle. It uses the formula

$$Area = \sqrt{s(s-a)(s-b)(s-c)} \qquad \text{where } s = \frac{a+b+c}{2}.$$

To check that the triangle exists, it checks that the sum of the two smaller sides is greater than or equal to the third side. (If they are equal, the triangle is degenerate.)
The code has been written as a procedure so that it can be tested easily for several choices of side-lengths. The first and last lines handle the procedure- aspect of the code, and should not cause confusion.
The first two lines of the procedure body (i.e., the second and third lines of the program) are sequence manipulations whose purpose is to assign `A`, `B`, `C` the same values as a, b, c, but sorted from lowest to highest. `Explode`(Q) is a multiple return value function that returns the first term, the second term, the third term, and so on, of a given sequence Q.

```
> TriangleArea := procedure(a, b, c)
procedure> Q := Sort([a, b, c]);
procedure> A, B, C := Explode(Q);
procedure> sumAB := A + B;
procedure> if sumAB lt C then
procedure|if> print "No triangle has these side-lengths.";
procedure|if> elif sumAB eq C then
procedure|if> print "Triangle is degenerate (straight line).";
procedure|if> print "Area is 0 square units.";
procedure|if> else
```

```
procedure|if> s := (A+B+C)/2;
procedure|if> print "Area is", Sqrt(s*(s-A)*(s-B)*(s-C)), "square units.";
procedure|if> end if;
procedure> end procedure;
```

The procedure `TriangleArea`(a, b, c) may be called by supplying suitable expressions for a, b and c. For example:

```
> TriangleArea(6, 8, 10);
Area is 24.000000000000000000000000000 square units.
> TriangleArea(10.2, 12, 14.5);
Area is 60.466393507054809296259993295 3 square units.
> TriangleArea(6, 20, 3);
No triangle has these side-lengths.
> TriangleArea(13, 9, 4);
Triangle is degenerate (straight line).
Area is 0 square units.
```

The figures in the output have a rather large number of digits because the arguments are elements of the real field (or coerced into it for the square root computation), which has default precision 30. This precision could be adjusted if required.

4.1.3 The case-statement

The **case**-statement is rather like the `elif` variety of the **if**-statement, but it is used when the action depends on which of several values is equal to a given value. The syntax of the **case**-statement is:

```
case expression
when expression, ... , expression:
   statements
when expression, ... , expression:
   statements
⋮
else
   statements
end if;
```

The `else` clause is optional.
To execute this statement, MAGMA evaluates the expression given after `case`. It successively compares the value of this expression with the values of the expressions given after each `when`. If it finds a match, MAGMA executes the statements following the corresponding `when`. If none of the values match, MAGMA executes the statements after `else`, or does nothing if the `else` clause is absent.
MAGMA executes at most one of the blocks of statements in a **case**-statement. If more than one of the when options contains an expression whose value equals the value of the case expression, MAGMA will only execute the statements corresponding to the first of these when options, starting from the top.

Example:
The `TriangleArea`(a, b, c) procedure constructed above may be modified to use a **case**-statement. The expression given after case is `Sign`$(A + B - C)$, since it is the sign of $A+B-C$ that governs which statements should be executed. The intrinsic function `Sign`(x) returns 1, 0 or -1 according to whether x is positive, zero or negative.

```
TriangleArea := procedure(a, b, c)
procedure> q := Sort([a, b, c]);
procedure> A, B, C := Explode(q);
procedure> case Sign(A + B - C):
procedure> when -1:
procedure> print "No triangle has these side-lengths.";
procedure> when 0:
procedure> print "Triangle is degenerate (straight line).";
procedure> print "Area is 0 square units.";
procedure> else
procedure> s := (A+B+C)/2;
procedure> print "Area is", Sqrt(s*(s-A)*(s-B)*(s-C)), "square units.";
procedure> end case;
procedure> end procedure;
```

Notice that since the `else` section of a **case**-statement is optional, the `else` and the following statement could have been written as another when option instead:

```
procedure> when 1:
procedure> s := (A+B+C)/2;
procedure> print "Area is", Sqrt(s*(s-A)*(s-B)*(s-C)), "square units.";
```

4.2 Iterative Statement

Iteration or **looping** is the repeated performance of a task that is syntactically constant but for which the data may change. The terms iteration or loop may also be applied to instances of the task's performance. The classic method of specifying iteration in computer languages is by means of iterative statements. These are compound statements, since they surround other statements providing the description of the task which is to be performed repeatedly. As part of this section we introduce two forms of iterative statements presented in MAGMA : the **while**-statement and the **for**-statement.

In most cases of iterative processes, the programmer intends the whole iteration to stop at the end of one of the loops. There are two ways of stating when this is to happen: either the iteration is over a predetermined finite domain D, and continues until some identifier has assumed each value of D exactly once in successive loops; or, less predictably, a Boolean expression is evaluated at the beginning or end of each loop, and the iteration process concludes when the value of this expression changes. Occasionally, it is more convenient for the iteration to conclude in the midst of one of the loops; this may be achieved by means of an interrupt that follows upon the change in value of some intermediate condition.

MAGMA's iterative statements provide for all these circumstances. In one of them, the **for**-statement, the iteration is over a domain, according to the first kind of termination condition. In the others the **while**-statement termination is dependent on the value of a Boolean expression that is evaluated respectively at the beginning and end of each loop. MAGMA also provides statements for interrupting the usual course of the iterative statement: the **break**-statement for immediate termination of the whole iterative statement.

It should be remembered that in most cases the choice of a method of iteration for an algorithm affects the time taken to develop the code much more than the execution time. Convenience for the programmer and legibility of the program are the most important factors in the decision, except when the computations are going to consume a significant amount of CPU time.

4.2.1 The while-statement

The syntax of the **while**-statement is

```
while condition do
   statements
end while;
```

where there must be at least one statement between `do` and `end while;` these statements constitute the task to be iterated. The termination condition for the iteration process is given by the Boolean expression following `while`. This expression is evaluated at the beginning of each loop, including the first loop. If it is **true**, then the loop is executed; otherwise, the execution of the whole **while**-statement finishes. Note that if the value of the Boolean expression is already **false** before the looping has started, then the statement body is never executed.

Example:
Consider the problem of calculating the factorial of n (MAGMA has an intrinsic function `Factorial`(n) that should be used in practice to find this value.) Initially, the value of n will be fixed at 20. The following lines of code, incorporating a **while**-statement, cause the value of 20! to be stored in the identifier *answer*:

```
> n := 20;
> c := n;
> answer := 1;
> while c gt 0 do
while> answer := answer * c;
while> c := c - 1;
while> end while;
```

Notice that a special prompt is provided during the time that the statement body of the **while**-statement is being entered, as a reminder that end while has not yet been typed.

After the iterative statement has terminated, the number 20! will be stored in *answer*. It may be compared with the value returned by `Factorial`(20):

```
> answer;
2432902008176640000
```

```
> Factorial(20);
2432902008176640000
> answer eq Factorial(20);
true
```

When an iterative statement is typed into MAGMA , if the system detects an error before the end of the whole statement has been reached, then the part of the statement already typed is lost. The workings of the code in the example above will now be explained in detail. The first statements this program executes are the assignments to n, c and *answer*. The reason for creating c as a copy of n is that the algorithm requires the value of n to be gradually decremented, but it may be undesirable to lose the original value of n. Next comes the **while**-statement. The first step in executing a **while**-statement is to evaluate the condition given between **while** and **do**. Since c is indeed greater than 0, MAGMA starts executing the statements between do and end while. On this first pass through the loop, answer is 1 and c is 20, so the line

```
while> answer := answer * c;
```

or, alternatively,

```
while> answer *:= c;
```

multiplies answer by c, making answer become 20. The other statement in the loop reduces c by 1. MAGMA now evaluates the **while**-condition again, and finds that it is still **true**, since $19 > 0$. Thus the loop statements are executed again. This time answer becomes 20×19 and c becomes 18. After several more iterations, execution of the loop reaches the stage when the value of c is 1. This time answer becomes $20 \times 19 \times \cdots \times 2 \times 1 = 20!$ and c becomes 0. Magma evaluates the **while**-condition yet again, but it is no longer **true**, because `0 gt 0` is **false**. Now execution of the program falls through to the line following `end while;` and answer has its final value of 20!.

4.2.2 The for-statement

The general structure of the **for**-statement is

```
for identifier in domain do
   statements
end for;
```

The loop identifier, following the word `for`, takes the elements in the domain as its successive values, and for each value of the identifier, the statements are executed. Typically they would include some reference to the loop identifier. The domain of the loop identifier should be an expression that evaluates to some iterable structure, that is, an iterable aggregate structure or a finite magma whose elements MAGMA is able to list. This domain is calculated when the execution of the **for**-statement begins, and cannot be changed during the iteration.

Example:
Consider the residue class ring $\mathbb{Z}_{18}$ of integers modulo 18. Some of the elements n of this ring are squares, in the sense that there exists an element m of the ring such that $m^2 = n$. One way to identify these squares is to test each element of the ring in turn, using the function `IsSquare`(n). The **for**-statement below does this, using the residue class ring as its domain:

```
> for n in ResidueClassRing(18) do
```

```
for> if IsSquare(n) then
for|if> print n;
for|if> end if;
for> end for;
0
1
4
7
9
10
13
16
```

To execute the **for**-statement above, MAGMA initially assigns to n the first element of $\mathbb{Z}_{18}$ delivered to it by the iteration mechanism for this ring. Then it performs everything between the `do` and the `end for;`, using this value of n. Next, n is assigned the second element of the ring and the body of the **for**-loop is executed again, with the new value of n. The process continues until the loop body has been executed for each element of the domain.

The **for**-construct is designed with a safeguard: the loop identifier is local to the loop. Effectively, it has no existence outside the loop. For instance, after the loop over $\mathbb{Z}_{18}$ has finished, it is not possible to obtain the final value that n had. Indeed, if there were an identifier called n that was assigned before the **for**-statement started, then when the loop had finished n would have its old value, not its final value in the loop; on the other hand, if there were no identifier n prior to the loop, then this would still be the case after the execution of the **for**-statement had finished. This scoping rule is designed to protect the user from destroying important data existing outside the loop.

Example:

```
> a := 134237402;
> A := AbelianGroup([2, 3]);
> for a in A do
for> print Order(a);
for> end for;
1
6
3
2
3
6
> a;
134237402
```

If the domain of a **for**-loop is a sequence (or a list or indexed set), the loop identifier will run through the elements of the domain in the standard indexing order. If it is a sequence (or a list or multiset) with repeated elements, then those elements will be assigned to the loop identifier the corresponding number of times.

Example:
The following statement prints each term r of a sequence of rational numbers, preceded by the floor $\lfloor r \rfloor$ of r (the greatest integer less than or equal to r):

```
> for j in [9/2, 1, 3/4, 2, 3/4, 3/4, 54/11] do
for> print Floor(j), j;
for> end for;
4 9/2
1 1
0 3/4
2 2
0 3/4
0 3/4
4 54/11
```

The output shows that the terms of the sequence are used in order, including the repeated terms.

If the domain of a **for**-statement is a sequence Q with repeated terms and it is undesirable to repeat the iteration for the same value of the loop identifier, then the **for**-statement should be given the enumerated set `Seqset`(Q) as its domain. This change can be important when the execution time for each iteration is non-trivial and the proportion of repeated terms is high.

4.2.3 Exiting an Iteration or Loop Quickly

Exiting the Current Iteration Quickly

It sometimes happens within the body of a loop that it is unnecessary to execute the rest of the statements in the loop body for the current iteration. This often happens when an iterative statement is searching for objects satisfying certain conditions; once it becomes established that a particular object does not satisfy these conditions, it should be possible to proceed immediately to the next object, in the next iteration. The MAGMA statement for this situation is

```
continue;
```

When MAGMA is executing a loop body and encounters this statement, it omits the rest of the statements in the loop body. Execution will proceed to the next iteration or, if all the iterations have been done, to the statement following the whole loop.

Example:
The program on 37 that prints the squares in $\mathbb{Z}_{18}$ may be modified as follows to use a **continue**-statement:

```
> for n in ResidueClassRing(18) do
for> if not IsSquare(n) then
for|if> continue;
for|if> end if;
for> print n;
for> end for;
0
```

```
1
4
7
9
10
13
16
```

In this case, there is only one statement left in the loop body after the **continue**-statement, so there is no real advantage to the modification. For larger loops, continue is useful because it helps avoid a tangle of incomplete **if**-statements.

Exiting the Whole Loop Quickly

MAGMA also has a statement

```
break;
```

which causes immediate exit from the entire iterative statement. This statement is useful in dealing with special cases, or in algorithms that are searching for a single value.

Example:

```
> for n in ResidueClassRing(18) do
for> print n;
for> if IsPrimitive(n) then
for|if> break;
for|if> end if;
for> end for;
0
1
2
3
4
5
```

Exits in Nested Loops

If one loop resides within another, it is said to be nested within the larger loop. When MAGMA encounters

```
continue;
```

or

```
break;
```

within a nested loop, it assumes that these statements refer to the innermost, smallest loop.

In the case of **for**-loops, it is possible to override this assumption by specifying the loop identifier of the relevant loop, after the word continue or break. For example, in the following nested **for**-loops:

```
for x in xdomain do
    ⋮
    for y in ydomain do
        ⋮
        breax x;
        ⋮
    end for;
    ⋮
end for;
```

the statement

```
break x;
```

refers to the outer loop, so the outer loop will be terminated. If the statement were

```
break;
```

then the inner loop would be terminated.

4.2.4 Iterations without Iterative Statements

MAGMA's set and sequence constructors, and related syntactic constructs, allow many kinds of iteration to be performed without the need for iterative statements.
The following table shows MAGMA's special methods for constructing an enumerated set or enumerated sequence S whose elements are integers in arithmetic progression.

MAGMA	Meaning
`{i..j}`	Set $\{i, i+1, \ldots, j\}$, where $i, j \in \mathbb{Z}$
`[i..j]`	Sequence $[i, i+1, \ldots, j]$, where $i, j \in \mathbb{Z}$
`{i..j by k}`	Set $\{i, i+k, i+2k \ldots, j\}$, where $i, j, k \in \mathbb{Z}$
`[i..j by k]`	Sequence $[i, i+k, i+2k \ldots, j]$, where $i, j, k \in \mathbb{Z}$

Notice that, if j is not of the form $i + mk$ for some integer m, then the progression continues as long as the range from i to j is not exceeded (that is, if $k > 0$ the last element in the progression will be the greatest integer of the form $i + nk$ that is less than j, and if $k < 0$ the last element will be the least integer of the form $i + nk$ that is greater than j).

Example:

```
> [20..7 by -3];
[ 20 .. 8 by -3 ]
> {x: x in $1};
{ 8, 11, 14, 17, 20 }
```

The universe of the result of an arithmetic progression constructor is always the ring of integers, but aggregates with other universes may be constructed from them, using the element-description method to be explained below.

Example:
The following expression returns the arithmetic sequence of rational numbers ranging from zero to one and incrementing by tenths:

```
> [x/10: x in [0..10]];
[ 0, 1/10, 1/5, 3/10, 2/5, 1/2, 3/5, 7/10, 4/5, 9/10, 1 ]
```

Example:
The following set constructor creates the set of integers of the form p^3, where p is a prime in the range $1 \leq p \leq 20$:

```
> PrimesCubed := { p^3 : p in [1..20] | IsPrime(p) };
> PrimesCubed;
{ 8, 27, 125, 343, 1331, 2197, 4913, 6859 }
```

Although this set is described statically, iteration takes place internally. MAGMA must iterate over the domain sequence $[1, \ldots, 20]$, and for each element p of the domain it must test its primality and then, for the primes only, include the value of p^3 in a set.

Chapter 5

Functions and Procedures

There are three kinds of functions and procedures in MAGMA . Firstly, there are the system **intrinsics**, which are the routines supplied in MAGMA as specialist implementations of algorithms for various categories. The other two kinds are created by the user. **User-defined functions and procedures** are designed for a particular application. They are defined during a MAGMA session, either from interactive input at the keyboard or when an input file is loaded. Lastly, there are **user intrinsics**, which are user-defined functions and procedures that are so generally applicable or important to the user that they have been placed in a **package**, using special syntax, in order to be compiled and treated like system intrinsics. This chapter explains how to invoke (call) all kinds of functions and procedures, and how to create user-defined functions and procedures.

Functions return one or more values, and procedures change the calling context. A function or procedure may have zero or more value arguments, and a procedure may also have zero or more reference arguments. Furthermore, a function or procedure may have zero or more parameters, with associated default values.

User-defined functions and procedures are created by means of an expression that evaluates to a function object or procedure object; this object is usually assigned to an identifier so it can be invoked easily. Recursion and forward declaration are permitted. There are two syntactic forms for user- defined functions: a **func**-constructor whose left side contains the formal arguments and whose right side contains expressions for the return values in terms of the arguments; and a more traditional form consisting of the formal arguments and then a body of statements, including a return statement with expressions for the return values. The syntax for user-defined procedures is similar: a **proc**-constructor whose left side contains the formal arguments and whose right side contains a procedure invocation in terms of the arguments; and a more traditional form consisting of the formal arguments and then a body of statements. Non-local identifiers are visible, but cannot be reassigned. For the syntactic forms involving a body of statements, the scope of identifiers is determined by a first textual use rule, though explicit local declaration is permitted as a precaution.

5.1 User-Defined Functions

In MAGMA, a user-defined function is created as the value of a function expression, that is, an expression which evaluates not to an integer, a sequence, or so on, but to a function. There are

two syntactic forms for function expressions in MAGMA.

5.1.1 Constructor Form of Function Expression

The **func**-constructor is the simplest form of function expression, and has the following syntax:

`func<` formal arguments | expressions for return values `>;`

On the left of the | symbol are comma-separated identifiers for the formal arguments of the function, and on the right are comma-separated expressions in terms of the formal arguments for the values that the function returns.
Typically, an identifier name is chosen for the function. If so, an assignment statement is used:

identifier `:= func<` formal arguments | expressions for return values `>;`

Example:
Suppose that a function is required to find the order of the general linear group, $GL(n, \mathbb{F}_q)$, for given values of n and q. The order of this group is equal to

$$\prod_{i=1}^{n}(q^n - q^{i-1})$$

so an expression for this value is

```
&*[q^n - q^(i-1) : i in [1..n]]
```

A function allows the user to gain access to this expression easily, without typing it each time that the order is required. The following line shows how to construct a function `orderGL(n, q)` which returns the value of this expression:

```
> orderGL := func< n, q | &*[q^n - q^(i-1) : i in [1..n]] >;
```

The value of the identifier `orderGL` is the function, and the formal arguments of the function are n and q. After `orderGL` has been defined as a function, it may be used to find the order of $GL(n, \mathbb{F}_q)$ for various n and q. For instance, the order of $GL(3, \mathbb{F}_4)$ may be printed:

```
> orderGL(3, 4);
   181440
```

and `ord32` may be assigned the order of $GL(3, \mathbb{F}_4)$:

```
> ord32 := orderGL(3, 2);
> // next line compares it with standard function Order
> print ord32 eq Order(GL(3, GF(2)));
true
```

If a function has to return more than one value, then expressions for each return value must be given on the right side of the **func**-constructor, separated by commas.

Example:
The following function `numden`(f) returns the numerator and denominator of a given fraction f, where f is an element of the rational field or a function field:

```
> numden := func< f | Numerator(f), Denominator(f) >;
```

When this function is called, the values are returned in the same order in which they are given in the constructor. The return values may be printed directly, or may be assigned to two identifiers:

```
> print numden(1178612/671674);
589306 335837
>
> P<x> := PolynomialRing(IntegerRing());
> F<y> := FieldOfFractions(P);
> n, d := numden( (y^7 - 4*y^2 + 509) / (12*y^3 + 5*y) );
> print n;
x^7 - 4*x^2 + 509
> print d;
12*x^3 + 5*x
```

5.1.2 Statement Form of Function Expression

In the `orderGL` example above, the return expression of the function was written directly in terms of the formal arguments. If a function is being defined for which it is impossible or awkward to do this, then a more general form of function expression should be used instead of the **func**-constructor. The statement form of function expression permits the return value(s) to be calculated using several intermediate statements. It has the syntax

`function` (formal arguments)
    statements
`end function;`

or

`function` name (formal arguments)
    statements
`end function;`

The body of the function may contain any number of statements, provided that when MAGMA is executing a call to this function, it will come to a line of the form

`return` expression, . . . , expression;

somewhere in the statement body. As soon as MAGMA encounters a **return**-statement like this, it will evaluate the expression(s) and return the result(s). Then the function call is finished, and any remaining lines in the function body will be ignored.

Example:
`orderGL` may be created using this form of function expression as follows:

```
> orderGL := function(n, q)
function> return &*[q^n - q^(i-1) : i in [1..n]];
function> end function;
```

These versions of the functions perform in the same way as those created using the **func**-constructor.

5.2 User-Defined Procedures

There are two ways to define a procedure. These methods correspond to the syntax for user-defined functions: the constructor form and the statement form. Both of these are expressions which evaluate to a procedure; the value of the expression is usually assigned immediately to an identifier.

5.2.1 Constructor Form of Procedure Expression

The constructor form of the procedure expression is discussed only briefly here, because it is rarely used. The **proc**-constructor has the following syntax:

`proc<` formal arguments | procedure call `>;`

On the left of the | symbol are comma-separated identifiers for the formal arguments (value and/or reference), and on the right is an procedure invocation in terms of the formal arguments. If the procedure is being assigned to an identifier, then the assignment statement will have the following form:

identifier `:= proc<` formal arguments | procedure call `>;`

The constructor form of procedure expression is rather limited in its applicability because the right side of the constructor has to be a single procedure call.

Example:
Suppose that a procedure `ShiftR`($\sim Q$) is required that modifies Q by shifting it one place to the right. `ShiftR` may be defined by invoking the system intrinsic procedure `Rotate`($\sim Q, n$) on the right side of a **proc**-constructor:

```
> ShiftR := proc< ~Q | Rotate(~Q, 1) >;
>
> V := VectorSpace(GF(5), 7);
> b := Eltseq(V.2);
> b;
[ 0, 1, 0, 0, 0, 0, 0 ]
> ShiftR(~b);
> b;
[ 0, 0, 1, 0, 0, 0, 0 ]
> print V!b;
```

```
(0 0 1 0 0 0 0)
```

5.2.2 Statement Form of Procedure Expression

The syntax of the statement form of a procedure expression is:

```
procedure (formal arguments)
    statements
end procedure;
```

or

```
procedure name (formal arguments)
    statements
end procedure;
```

where each formal argument may be a reference argument or a value argument; every reference argument must be preceded by a tilde.

Example:
The procedure `ShiftR` may be created in statement form as follows:

```
> ShiftR := procedure(~Q)
> Rotate(~Q, 1);
> end procedure;
```

A common use for a procedure is to print output.

Example:
The following trivial example has no arguments, and always performs the same task, printing a line of sixty asterisks:

```
> SeparatingLine := procedure()
> print "*"^60;
> end procedure;
> SeparatingLine();
************************************************************
```

A more flexible procedure would be one with two value arguments c and n that allow the procedure call to specify which character c is to be printed and the number n of such characters. It could be used as a separating line for part of a printing routine.

```
> SeparatingLine := procedure(c, n)
> print c^n;
> end procedure;
> SeparatingLine("#", 35);
###################################
```

5.3 Local and Non-Local Identifiers

At this point it may seem that there is potential for great confusion between identifiers of the same name residing inside and outside of a function/procedure. MAGMA has a simple principle known as the *first textual* use rule to prevent problems like this. Any identifier which is a formal argument, or whose first appearance in the text of the function expression is for the purpose of having a value assigned to it, is local to the function. This means that if there are identifiers with the same name existing outside the function, they are not changed by a function call, and on the other hand, after the function call has finished execution, the function's identifiers effectively disappear and those outside the function reappear.

Example:

```
> a := 17; b := 62; c := 54;
> multip := function(a, b)
> c := a * b;
> return c;
> end function;
>
> multip(8, 9);
72
> a, b, c;
17 62 54
```

Although the formal identifiers a and b are given the values 8 and 9 within the scope of the function, the values of the identifiers a and b outside the function do not change. Similarly, although the identifier c inside the function is given the value 72, the value of the identifier c outside the function does not change. Therefore anyone can use the function `multip` by knowing merely that it returns the product of its two arguments. As is appropriate, the internal workings of the function are irrelevant to its use.

Identifiers in functions are not always local. If the first time that an identifier appears is when it is referred to in an expression rather than when it is assigned, then Magma looks for its value outside the function.

Example:

```
> a := 42;
> function test(b)
function> c := a*(3-b);
function> return c^2+a;
function> end function;
>
> test(6);
15918
```

The first textual use of a in the function body of `test` is in the expression $a*(3-b)$. Therefore a is a non-local identifier, and MAGMA will use the value which a has been assigned outside

the function. The result is that silly is defined exactly as if it had been:

```
> function test(b)
function> c := 42*(3-b);
function> return c^2+42;
function> end function;
```

Note that if the value of a is now changed, the function `test` will remain as it was before, although its definition included the use of a. Only the current values of identifiers are relevant when a function is being defined:

```
> a := 100000;
> print test(6);
15918
```

It is generally unwise to use non-local identifiers within a function. The danger is that the function could be defined at a time when its non-local identifiers are set incorrectly; this can easily happen, especially when a function is in the development stage. A non-local identifier should only be used in a function definition when it can be guaranteed that its value will be as expected.

Part II

Algebraic Structures

Chapter 6

Rings and Fields

This chapter consists of an overview of the main categories of commutative rings in Magma, and the operations that may be performed on them. Fields are included in the discussion, as special cases of rings.

The various categories of commutative rings cover a very wide spectrum, and some of the most important objects in computer algebra are found here. Subsequent chapters explain in detail how to use the most important particular types of commutative rings, such as the ring of integers $\mathbb{Z}$, the field of rational numbers $\mathbb{Q}$, residue class rings $\mathbb{Z}_m$, real and complex fields, univariate and multivariate polynomial rings, power series rings, finite fields, number fields and function fields. Some important types of non-commutative rings (such as matrix rings) appear in the part on algebras. Their category names (like `AlgMat`) start with `Alg`, whereas the names of ring categories usually start with `Rng`, or with `Fld` if only fields are involved.

In this Chapter we describe some of the general principles for the construction and use of rings and fields, and some principles underlying the design and organization of this diverse area.

6.1 Creating Rings and Fields

Two different families of ring creation functions can be distinguished. The first family is that of standard functions to create rings (and fields) directly. The second family consists of constructors and functions that create new rings from existing ones.

6.1.1 Standard Creation Functions

Many rings and fields in Magma may be created directly using standard functions. Thus $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, finite fields $\mathbb{F}_q$ and residue class rings $\mathbb{Z}_m$ may be created using an intrinsic function with the appropriate arguments.

Example:

```
> Q := RationalField();
> Q;
Rational Field
>
```

```
> Z12 := ResidueClassRing(12);
> Z12;
Residue class ring of integers modulo 12
>
> G := FiniteField(5, 2);
> G;
Finite field of size 5^2
>
> Category(Q), Category(Z12), Category(G);
FldRat RngIntRes FldFin
```

The meaning of the arguments in the above will be clear. In certain cases, such as the finite field example, different numbers of arguments are allowed: `FiniteField`(25) is synonymous to `FiniteField`$(5, 2)$.

The following table shows some functions for standard rings.

`IntegerRing`	`RationalField`	`RealField`
`ComplexField`	`FiniteField`	`NumberField`
`PolynomialRing`	`FunctionField`	`PowerSeriesRing`

Example:
Here the construction of the number field $\mathbb{Q}(\sqrt[3]{2})$.

```
> R<t> := PolynomialRing(Integers());
> K<k> := NumberField(t^3-2);
```

6.1.2 Recursive Ring Constructions

The construction of new rings from old ones occurs in various guises. Firstly it may occur that the ring construction involves a 'coefficient ring' as parameter. We saw an example of that already in the polynomial ring definition above.

Example:
Here are two more examples, of $\mathbb{Q}(\sqrt[3]{2})[x, y]$ and $\mathbb{F}_5[[z]]$ respectively.

```
> R<t> := PolynomialRing(Integers());
> K<k> := NumberField(t^3-2);
> Q<x,y>:=PolynomialRing(K,2);
> P<t> := PowerSeriesRing(FiniteField(5));
```

The user should beware of possible problems in using recursively defined rings and algebras over 'approximate' ground rings, such as the real numbers. Numerical instabilities can lead to unexpected problems, for example in dealing with matrices over real or complex numbers.

6.1.3 Ring Construction

A second form of constructing rings from existing ones is given by the constructors `sub`, `ext`, and `quo`, for subrings, extensions, and quotient rings. These constructors are not available for

all rings, since in some cases there are no algorithms to compute the new structures effectively. Subrings, extensions, quotients are rings in their own right, and are the parents of the elements that belong to them. Although the syntax for these constructors is fairly uniform, they return objects of different kinds.

The `sub` constructor for a ring R has the form

`sub<` R | list of ring generators `>`

It returns two values: the subring S of R generated by the elements listed, and the embedding ring homomorphism from S to R. The ring S will be in the same category as R.

Example:
The following line constructs the subring of a finite field of 625 elements, generated by the element g^{26}:

```
> G<g> := FiniteField(625);
> F<f>, m := sub< G | g^26 >;
> m(f^2);
g^52
```

The syntax of the **ideal** constructor is similar to that of the `sub` constructor. For some categories of rings the ideals form a separate category in MAGMA .

Example:
Here is an ideal in a univariate polynomial ring:

```
> R<x> := PolynomialRing(RationalField());
> f := x^3 + 1/2*x^2 - 3*x - 3/2;
> I := ideal< R | f >;
> I;
Ideal of Univariate Polynomial Ring in x over Rational Field
generated by x^3 + 1/2*x^2 - 3*x - 3/2
```

The **quo** constructor creates the quotient of a ring R by an ideal of R. It returns two values: the quotient, and the natural homomorphism from the ring to the quotient.

Example:
Continuing the previous example,

```
> Q<y>, h := quo< R | I >;
> Q, h;
Univariate Quotient Polynomial Algebra in y over Rational Field
with modulus y^3 + 1/2*y^2 - 3*y - 3/2
Mapping from: RngUPol: R to RngUPolRes: Q
> h(x), h(f);
y
0
```

The ideal may be specified explicitly as an ideal, as above, or in terms of its generators:

```
> Q2<z> := quo< R | f >;
> Q2;
Univariate Quotient Polynomial Algebra in z over Rational Field
with modulus z^3 + 1/2*z^2 - 3*z - 3/2
```

The quotient is generally an object of another category than that of the ring; which category depends on R.

The `ext` constructor has a limited applicability in categories of fields and for transcendental extensions: it may be used to extend finite fields or number fields, and to create polynomial rings.

For transcendental extensions there are two versions:

`ext<` R `|` `>`

is the same as `PolynomialRing`(R), returning a univariate polynomial ring over R, and:

`ext<` R, n `|` `>`

is the same as `PolynomialRing`(R, n), returning a multivariate polynomial ring of rank n over R.

Example:

```
> P<q,r,s,t> := ext< Z, 7 | >;
> P;
Polynomial ring of rank 7 over Integer Ring Lexicographical Order
Variables: q, r, s, t, $.5, $.6, $.7
```

The main constructor for algebraic extensions has the form

`ext<` F `|` f `>`

where f is a monic irreducible polynomial over F. The constructor returns an extension E of F by a root of f.

Example:

```
> K<w> := GF(8);
> PK<X> := PolynomialRing(K);
> f := X^2 + w*X + 1;
> IsIrreducible(f);
true
> E<alpha> := ext< K | f >;
> E;
Finite field of size 2^6
> IsZero(Evaluate(f, alpha));
true
```

6.2 Operation on Ring Elements

Methods for creating elements of a ring R depend on the category of R, but it generally involves one of the following constructions: the coercion into R of an element or a sequence of elements of a ring related to R; an expression in the generators of the ring, using the arithmetic operations; or the construction of some special element of R.

Example:

```
> Z12 := Integers(12);
> Z12 ! 57;
9
>
> R<x> := PolynomialRing(RationalField());
> p := R ! [5/4, 0, 13/7, -1, 2];
> p;
2*x^4 - x^3 + 13/7*x^2 + 5/4
> R<s, t> := PolynomialRing(FiniteField(5), 2);
> s*t^5 + 4*t + 7;
s*t^5 + 4*t + 7
```

The following table shows how to create some particular elements of the ring R.

MAGMA	Meaning
`R.i`	i-th generator of ring R
`Zero(R)`	Additive identity of ring R
`One(R)`	Multiplicative identity of unitary ring R
`Representative(R)`, `Rep(R)`	A representative element of ring R
`Random(R)`	A random element of finite ring R
`Parent(x)`	Parent ring of ring element x
`Category(x)`, `Type(x)`	Category to which ring element x belongs

We will briefly discuss the use of the angle brackets on the left hand side of an assignment here. Angle brackets are used to give names to 'indeterminates' (as in the above example), or elements that are 'generators' in a more general sense, for example in number fields or finite fields. ('Generator' here means generator of the new ring that is being constructed as an algebra over the ring it is defined over.) It is very important to know that the effect of:

```
> R<s, t> := PolynomialRing(FiniteField(5), 2);
```

instead of

```
> R := PolynomialRing(FiniteField(5), 2);
```

is twofold. In the first place the 'names' placed between angle brackets (separated by comma's) are used as strings in MAGMA output: indeed, elements of R will be printed as polynomials in which 's' and 't' are the indeterminates. Simultaneously, however, (two) identifiers (s and t) will be assigned values, being the first indeterminates or generators in the ring that is being constructed. This makes it possible from here on to define elements (of R in the above example) using the identifiers (s and t).

Example:
That the two operations are indeed distinct can be seen from the following lines, where s and t as identifiers are reassigned, but the output of polynomials is not affected:

```
> R<s, t> := PolynomialRing(FiniteField(5), 2);
> f := s^3+t^2;
> f;
s^3 + t^2
> s := -2; t := 1; > s^3 + t^2;
-7
> f;
s^3 + t^2
```

If identifiers have been reassigned, or if the angle brackets were not used in the first place, it is still possible to do the assignment afterwards. For this the 'dot' notation is used. This makes it possible to get hold of the i-th generator of a ring, via by `R.i`.

Example:
Continuing the above example

```
> u := R.1; v := R.2;
> u^2*v;
s^2*t
```

Indeed, the output has not changed! It is also possible to change the string used in printing, afterwards, with the `AssignNames` procedure.

Example:
> AssignNames(R, ["blue", "red"]); > u + v; blue + red

The second argument of `AssignNames` is a sequence of strings. Like the number of names in angle brackets the length of that sequence is allowed to be smaller but not larger than the number of generators. If the not all generators are given new names, Magma resorts to the default option, which is `$.i` for the i-th generator.

Example:

```
> AssignNames(~R, ["X"]);
> u + v;
X + $.2
```

Given a ring or algebra R, `Zero`(R) is the additive identity and `One`(R) is the multiplicative identity. These elements can also be obtained by coercion of the integers 0 and 1.

Example:

```
> M := MatrixRing(Integers(12), 3);
> One(M);
[1 0 0]
[0 1 0]
[0 0 1]
```

```
> M!0;
[0 0 0]
[0 0 0]
[0 0 0]
```

Once some ring elements have been created, others can be constructed from these using the arithmetic operations in the ring. For convenience we list the standard operators in the following table.

MAGMA	Meaning
`x+y`	Sum of elements x and y
`-x`	Additive inverse of x
`x-y`	Difference of elements x and y
`n*x`	x added to itself n times
`x*y`	Product of elements x and y
`x^-1`	Multiplicative inverse of unit x
`x^n`	x^n, where n is an integer (n may be negative only if x is a unit)
`x/y`	xy^{-1}, where y is a unit

Of course not every element has an inverse, and therefore there are also restrictions on the second argument for `/`. Note that we already saw how `/` is sometimes used to create elements of a field of fractions (for example in the case of integers); in such circumstances it is useful also to have exact division, where the result is in the ring itself. The operator `div` is provided for that.

Example:

```
> q := 6/3;
> q, Parent(q);
2 Rational Field
> z := 6 div 3;
> z, Parent(z);
2 Integer Ring
```

There are many Boolean operators and functions for ring elements. Firstly, there are the standard equality-testing operators, `eq` and `ne`. Next, there are the order operators such as `gt`. They are only available for rings which have a total ordering defined on their elements; these rings are the integer ring, the rational field and the real field. The other Boolean operations on ring elements are the functions beginning with `Is`, as listed in following table.

MAGMA	Meaning
`IsZero(x)`	**true** if x is the additive identity of its parent ring
`IsOne(x)`	**true** if x is the multiplicative identity of its parent ring
`IsMinusOne(x)`	**true** if x is the -1 of its parent ring
`IsUnit(x)`	**true** if $xy = 1$ for some y in the parent ring
`IsNilpotent(x)`	**true** if $xq = 0$ for some $q \in \mathbb{Z}_+$; if **true**, also returns the smallest such q (the index of nilpotence)
`IsZeroDivisor(x)`	**true** if $x = 0$ and $xy = 0$ for some $y = 0$
`IsRegular(x)`	**true** if x is not a zero divisor
`IsIdempotent(x)`	**true** if $x^2 = x$
`IsIrreducible(x)`	**true** if x (in an integral domain) is not a unit, and if whenever $x = ab$, then a or b is a unit
`IsPrime(x)`	**true** if x (in an integral domain) is not 0 nor a unit, and if $x\|ab$, then $x\|a$ or $x\|b$

6.2.1 Testing Properties of a Ring

There are many Boolean functions which test properties of a ring (or field) R, listed in the following table.

MAGMA	Meaning
`IsField(R)`	**true** if R is a commutative division ring
`IsOrdered(R)`	**true** if 'greater than' is defined in R
`IsEuclideanDomain(R)`	**true** if R is a domain and has a known Euclidean norm function
`IsEuclideanRing(R)`	**true** if R has a known Euclidean norm function
`IsPrincipalIdealDomain(R)`, `IsPID(R)`	**true** if R is a domain and every ideal of R has the form aR for some $a \in R$
`IsPrincipalIdealRing(R)`, `IsPIR(R)`	**true** if every ideal of R has the form aR for some $a \in R$
`IsUniqueFactorizationDomain(R)`, `IsUFD(R)`	**true** if R is a domain and factorization is unique
`IsIntegralDomain(R)`, `IsDomain(R)`	**true** if R is commutative, has a $1 \neq 0$, and has no zero divisors
`IsCommutative(R)`	**true** if R is commutative
`IsDivisionRing(R)`	**true** if R has a $1 \neq 0$, and every $a \in R$ except 0 has an inverse
`IsUnitary(R)`	**true** if R has a 1
`IsFinite(R)`	**true** if R has a finite number of elements; if **true**, also returns the cardinality `#R` of R

The return value depends on the mathematical properties of R, not the category to which R belongs.

Example:

```
> Z9 := ResidueClassRing(9);
```

```
> IsField(Z9);
false
> Z7 := ResidueClassRing(7);
> IsField(Z7);
true
> IsPrincipalIdealDomain(Z7);
true
```

If there is no appropriate algorithm (implemented) for determining the answer, MAGMA will give an error message.

Even if a ring R returns **true** to one of these Boolean functions, operations employing this property may not be available if the property does not hold for all rings in the category of R. This follows from the MAGMA-wide principle that the operations that apply to an object are given by the category of definition. For instance, $\mathbb{Z}_7$ is not a member of a field category, so it is not possible to construct a vector space over it, even though MAGMA can tell that $\mathbb{Z}_7$ satisfies the axioms of a field. However, it is possible to construct a vector space over `FiniteField`(7), which is isomorphic to $\mathbb{Z}_7$. The user must explicitly create the structure as a field if MAGMA is expected to regard it as a field.

6.3 Operations on Various Kinds of Rings

The following table lists some of the additional operations on rings.

MAGMA	Meaning
`Characteristic(R)`	Smallest positive integer m such that $mr = 0$ for every $r \in R$, or 0 if such an m does not exist
`PrimeRing(R)`	For a field F, returns $\mathbb{F}_p$ if the characteristic p of F is positive, else the rational field
`R+S`	Given subrings R and S of the same ring, return ring $\{r + s : r \in R, s \in S\}$; generators will be the sums of the generators of R and S
`R*S`	Given subrings R and S of the same ring, return ring $\{rs : r \in R, s \in S\}$; generators will be the products of the generators of R and S
`R meet S`	Given subrings R and S of the same ring, return ring $R \cap S$

These operations are in theory available for all kinds of rings. In practice, however, there are categories of rings, or certain rings within a category, for which certain operations are well-defined but cannot be easily computed. MAGMA will give an error message if such operations are attempted on these rings.

Chapter 7

Polynomial Rings

Polynomials in MAGMA may have one or several indeterminates, and may be defined over any ring, even another polynomial ring. MAGMA has two types of polynomial rings: **univariate** and **multivariate**. Univariate polynomial rings always have one indeterminate, while multivariate polynomial rings may have more than one indeterminate. The two types of polynomial rings have quite different properties and features. Consequently, the two types of polynomial rings are implemented with different categories and although they share many simple operations, the major algorithms are quite distinct. Most functions in other areas of MAGMA which happen to deal with polynomials take and return univariate polynomials. For example, univariate polynomials are universal in computations with finite fields, as they arise in computations of minimal polynomials, field extensions, and embeddings. In linear algebra and module theory, functions like `CharacteristicPolynomial` return univariate polynomials. In contrast, multivariate polynomials usually only arise as elements of ideals of multivariate polynomial rings in the area of Commutative Algebra.

Note that a multivariate polynomial ring with one indeterminate is not the same as a univariate polynomial ring over the same coefficient ring. The term "univariate" does not just mean "one variable" but implies a type of polynomial ring with quite different properties (and representation). It is almost always best to use the univariate and multivariate facilities in the obvious way, rather than to mimic a multivariate ring by creating univariate polynomial rings with univariate polynomial rings as their coefficient rings, or to construct single-indeterminate multivariate polynomial rings.

7.1 Univariate Polynomial Rings

A univariate polynomial ring $P = R[x]$ over the coefficient ring R always has one indeterminate x and a univariate polynomial f of P is a sum of the form $\sum_{i=0}^{n} c_i x^i$, with $c_i \in R$ for each i. Such a polynomial f is represented internally as the coefficient vector $[c_0, \ldots, c_n]$. This is the most efficient way in general for representing univariate polynomials.

The operations available on univariate polynomials include arithmetic, computation of derivatives, evaluation for a particular value of the indeterminate, and division to find quotient and remainder. For certain coefficient rings there are functions for greatest common divisors, factorization, and resultants.

The category of univariate polynomial rings is `RngUPol`.

7.1.1 Constructing Polynomial Rings

The function to create a univariate polynomial ring is `PolynomialRing`(R), where R is the ring from which the coefficients are taken.

Example:
The following assignment defines the polynomial ring P over the integers in the indeterminate x:

```
> P<x> := PolynomialRing(IntegerRing());
> print P;
Univariate Polynomial Ring in x over Integer Ring
```

It is customary to supply the indeterminate's name by means of generator assignment (within angle brackets on the left of the assignment statement), so that it becomes an identifier and a printname. Otherwise, the indeterminate is referred to as `P.1`, since it is the only (first) indeterminate of P.

The functions `CoefficientRing`(P) and `Rank`(P) return the coefficient ring R of a polynomial ring P, and the number of indeterminates which P has (always 1 for univariate polynomial rings).

7.1.2 Creating Polynomials

There are several ways of creating elements of an univariate polynomial ring. The most straightforward is to follow the mathematical representation involving powers of the indeterminate.

Example:
The next line shows how to create the polynomial $f = x^4 - 5x^2 + 3x + 6$ as a member of P:

```
> f := x^4 - 5*x^2 + 3*x + 6;
> f;
x^4 - 5*x^2 + 3*x + 6
```

For constant polynomials, there is a slight complication, because the element of the coefficient ring has to be coerced into the polynomial ring.

Example:
The following assignment creates the polynomial $g = 7$:

```
> g := P!7;
```

There is another way of creating univariate polynomials, in which every coefficient is listed in order from the constant term to the leading coefficient. The coefficients may either be placed on the right side of an `elt` constructor, or in a sequence which is coerced into the ring.

Example:
The polynomial $h = 8x^7 + 5x^6 + 9x^5 - 2x^4 + x^3 - x + 3$ in P can be created as follows:

```
> h := P![3, -1, 0, 1, -2, 9, 5, 8];
> h;
```

```
8*x^7 + 5*x^6 + 9*x^5 - 2*x^4 + x^3 - x + 3
```

or alternatively as

```
> h := elt< P | 3, -1, 0, 1, -2, 9, 5, 8 >;
```

The coefficient-list method may be preferable when most of the coefficients are non-zero, whereas the method of expressing the polynomial in the inde- terminates is preferable when the degree is large but only a few coefficients are non-zero.

7.1.3 Operations on Univariate Polynomials

The following table lists some operations on univariate polynomials.

MAGMA	Meaning
`Coefficients(f)`	Sequence of coefficients of f in ascending order
`Coefficient(f, k)`	Coefficient of k-th power of indeterminate of f
`MonomialCoefficient(f, m)`	Coefficient that the monomial x^m has as a term of f
`Degree(f)`	Degree of f
`Terms(f)`	The non-zero terms of f as a sequence
`LeadingTerm(f)`	Term with highest occurring indeterminate
`LeadingCoefficient(f)`	Coefficient of leading term of f
`TrailingTerm(f)`	Term with lowest occurring indeterminate
`TrailingCoefficient(f)`	Coefficient of trailing term
`Reductum(f)`	f minus the leading term
`Derivative(f)`	Derivative of univariate f
`Derivative(f, n)`	n-th derivative of univariate f
`Evaluate(f, r)`	Value of univariate f when r is substituted for indeterminate; if r is in coefficient ring R, result will be in R, else (if possible) result will be in parent of r
`Quotrem(f,g)`	Given polynomials f and g, return polynomials q and r such that $f = qg + r$ and the degree of r is strictly less than the degree of g
`f div g`	q as described in Quotrem
`f mod g`	r as described in Quotrem

Example:

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^4 - 5*x^2 + 3*x + 6;
> Derivative(f);
4*x^3 - 10*x + 3
> Evaluate(f, 5);
521
> print Coefficient(f, 1);
3
```

Notice that `Evaluate` for univariate polynomials is able to accept some substitution values that do not belong to the coefficient ring.

Example:

```
> MR := MatrixRing(IntegerRing(), 3);
> m := MR ! [3, 6, 1, 34, 1, 5, 2, 0, 3];
> Evaluate(f, m);
[49101 10410  9255]
[61340 45161 16665]
[ 4410  2820  1161]
```

The zero polynomial is a special case. Mathematically, its degree is $-\infty$, but Magma deems its degree to be -1. Since it has no coefficients and no terms, the user should take care when writing routines that call these functions, including special treatment for the zero polynomial case as appropriate.

7.1.4 Factorization and Root-Finding

Factorization

The function `Factorization`(f) factorizes the univariate polynomial f into powers of irreducible factors.

Example:

```
> P<x> := PolynomialRing(IntegerRing());
> f := x^20-1;
> Factorization(f);
[
    <x - 1, 1>,
    <x + 1, 1>,
    <x^2 + 1, 1>,
    <x^4 - x^3 + x^2 - x + 1, 1>,
    <x^4 + x^3 + x^2 + x + 1, 1>,
    <x^8 - x^6 + x^4 - x^2 + 1, 1>
]
```

This function is available when the coefficient ring is $\mathbb{Z}$, $\mathbb{Q}$, a finite field, a residue ring modulo p, an algebraic number field $\mathbb{Q}(\alpha)$, or a rational function field over any of the above rings.
The first return value of `Factorization` is a sequence of 2-tuples whose first component is an irreducible factor and whose second component is the multiplicity of that factor. Like the `Factorization` function for integers, the `Factorization` function for polynomials has a second return value, the unit u of the coefficient ring such that $f = ug$, where g is the monic polynomial associated with f.

Example:
The following statement factorizes $-5x^{11} + 25x^{10} + x^8 - 5x^7 - 7x^6 + 31x^5 + 20x^4$ over the integers:

```
> P<x> := PolynomialAlgebra(IntegerRing());
> a, b := Factorization(-5*x^11 + 25*x^10 + x^8 - 5*x^7
> - 7*x^6 + 31*x^5 + 20*x^4);
```

```
> a,b;
[
    <x - 5, 1>,
    <x, 4>,
    <5*x^6 - x^3 + 7*x + 4, 1>
]
-1
```

Therefore the factorization of the polynomial is $-(x-5)x^4(5x^6 - x^3 + 7x + 4)$.

The function `IsIrreducible`(f) returns true if the polynomial f is irreducible.

Roots of Univariate Polynomials

Let f be a univariate polynomial over the ring R, where R may be $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{F}_q$ or $\mathbb{Z}_p$. A root of f is an object r such that when r is substituted for the indeterminate of f, the result is 0; unless otherwise stated, it is understood that r is an element of R. The function `Roots`(f) returns all the roots of f in R, in a format similar to a factorization sequence, that is, as a sequence of tuples $< r_i, m_i >$ where r_i is a root and m_i is its multiplicity.

Example:
Let f be the polynomial shown below. Its roots in $\mathbb{Q}$ are 3 (with multiplicity 2) and $-\frac{7}{2}$ (with multiplicity 1):

```
> PQ<x> := PolynomialRing(RationalField());
> f := (x - 3)^2*(x^2 + 9)*(x + 7/2);
> print Roots(f);
[ <3, 2>, <-7/2, 1> ]
```

It is also possible to find the roots of f lying in another ring S, if the coefficients of f can be coerced automatically into S (e.g., if R is a subring of S), by means of the function `Roots`(f, S).

Example:
Continuing the example:

```
>  Roots(f, ComplexField());
[ <-3.5000000000000000000000000000000, 1>, <-3.0000000000000000000000000000000*$.1,
1>, <3.0000000000000000000000000000000*$.1, 1>, <3.0000000000000000000000000000000,
2> ]
```

The output above indicates that the roots of f lying in the complex field are -3.5 (multiplicity 1), $-3i$ (multiplicity 1), $3i$ (multiplicity 1), and 3 (multiplicity 2).

7.2 Multivariate Polynomial Rings

Multivariate polynomial rings and their ideals deal with the subject of Com- mutative Algebra which has a much larger theory than that associated with univariate polynomial rings and their ideals. Consequently, the two types of polynomial rings are implemented with different categories and although they share many simple operations, the major algorithms are quite distinct.

Multivariate polynomials themselves are quite different from univariate polynomials in that they are linearly ordered sums of coefficient-monomial pairs. Furthermore, the orderings on the monomials within the polynomials play an extremely significant role.

The category of multivariate polynomial rings is `RngMPol`.

Let P be the multivariate polynomial ring $R[x_1, \ldots, x_n]$ of rank n over the coefficient ring R. A monomial (or power product) of P is a product of powers of the variables of P; that is, an expression of the form $x^{e_1} \cdots x^{e_n}$ with $e_i \geq 0$ for $1 \leq i \leq n$. A term of P is a product cm where c is a coefficient from R and m is a monomial of P.

Note that a multivariate polynomial ring of rank 1 over the coefficient ring R is not the same as the univariate polynomial ring over R. Usually, if one wishes to compute with univariate polynomials then the univariate polynomial ring should be used (particularly since there are many functions applicable to univariate polynomials which would be useful), and the multivariate polynomial ring of rank 1 should only be used when it occurs, say, as the base case of a recursive algorithm which computes with multivariate polynomial rings.

7.2.1 Polynomial Creation and Access

The only way to create a multivariate polynomial in MAGMA is to use an expression in the indeterminates, or to coerce an element of the coefficient ring into the polynomial ring to create a constant polynomial.

Example:
The following lines create the element $c = 4s^3 + 20s^2t + 19st^4 + 8t + 7$ in the polynomial ring $P = \mathbb{Q}[s, t]$:

```
> P<s, t> := PolynomialRing(RationalField(), 2);
> c := 20*s^2*t + 4*s^3 - 12*s*t^4 + 8*t - 24;
> c;
4*s^3 + 20*s^2*t + 19*s*t^4 + 8*t + 7
```

MAGMA provides a large number of functions for accessing multivariate polynomials. These functions are quite different from those applicable to univariate polynomials because the multivariate polynomials have quite a different structure as linearly ordered sums. Since a multivariate polynomial f is just a linearly ordered sum of coefficient-monomial pairs, f can be viewed in an *absolute* manner as two parallel lists: the list of base coefficients (from the coefficient ring) and the list of monomials. The following table lists the functions for accessing polynomials in the absolute manner.

MAGMA	Meaning
`Coefficients(f)`	Coefficients of f as a sequence
`LeadingCoefficient(f)`	Leading coefficient of f
`TrailingCoefficient(f)`	Trailing coefficient of f
`MonomialCoefficient(f, m)`	Coefficient of monomial m in f
`Monomials(f)`	Monomials of f as a sequence
`LeadingMonomial(f)`	Leading monomial of f
`Terms(f)`	Terms of f as a sequence
`LeadingTerm(f)`	Leading term of f
`TrailingTerm(f)`	Trailing term of f
`TotalDegree(f)`	Total degree of f (maximum of total degrees of monomials of f)
`LeadingTotalDegree(f)`	Total degree of leading monomial of f

For a fixed variable v, a multivariate polynomial f may be viewed recursively as a univariate polynomial with respect to variable v. That is, f may be considered as a linearly ordered sum whose terms are of the form $c_i v_i$ where c_i is the coefficient of the power v_i, which is still in general a polynomial (not containing v though, of course). MAGMA provides functions to access f considered in this way. For each function the variable v can be specified in two ways: as the integer $i \in \{1 \dots n\}$ (where n is the rank of the ring) which is the variable number which corresponds to v, or as the variable v itself (lying in the polynomial ring). The following table lists the functions for accessing polynomials in the recursive manner. The functions are only listed in the form in which the variable number i is given; for each function there is also a version which takes the actual variable v instead of the number i.

MAGMA	Meaning
`Coefficients(f, i)`	(Recursive) coefficients of f w.r.t. variable number i as a sequence
`Coefficient(f, i, k)`	k-th (recursive) coefficient of f w.r.t. variable number i as a polynomial
`LeadingCoefficient(f, i)`	(Recursive) leading coefficient of f w.r.t. variable number i
`TrailingCoefficient(f, i)`	(Recursive) trailing coefficient of f w.r.t. variable number i
`Terms(f,i)`	(Recursive) terms of f w.r.t. variable number i as a sequence
`Term(f,i,k)`	(Recursive) k-th term of f w.r.t. variable number i
`LeadingTerm(f, i, k)`	(Recursive) leading term of f w.r.t. variable number i
`TrailingTerm(f, i, k)`	(Recursive) leading term of f w.r.t. variable number i
`Degree(f,i)`	(Recursive) degree of f w.r.t. variable number i

Example:
The following example demonstrates the differences between the absolute and recursive access

functions for a polynomial ring with lex monomial ordering:

```
> P<x, y, z> := PolynomialRing(IntegerRing(), 3);
> f := 2*x^3*y*z + 4*y*z^3 + 8*z + 3;
> f;
2*x^3*y*z + 4*y*z^3 + 8*z + 3
> Coefficients(f);
[ 2, 4, 8, 3 ]
> Monomials(f);
[
    x^3*y*z,
    y*z^3,
    z,
    1
]
> Terms(f);
[
    2*x^3*y*z,
    4*y*z^3,
    8*z,
    3
]
> TotalDegree(f);
5
> LeadingMonomial(f);
x^3*y*z
> Degree(f, x);
3
> Coefficients(f, x);
[
    4*y*z^3 + 8*z + 3,
    0,
    0,
    2*y*z
]
> Coefficient(f, x, 0);
4*y*z^3 + 8*z + 3
```

7.2.2 Factorization

MAGMA incorporates powerful algorithms for factoring multivariate polynomials over various coefficient rings. Currently, the available coefficient rings are: the integer ring $\mathbb{Z}$; the rational field $\mathbb{Q}$; finite fields; cyclotomic, quadratic, and general number fields; and rational function fields or polynomial rings over any of the above.

The factorization algorithms for multivariate polynomials are generally much harder than the corresponding ones for univariate polynomials. Indeed, the easiest coefficient rings for univariate

polynomials are the finite fields, while for multivariate polynomials the finite fields become the most difficult coefficient rings!

The function `Factorization`(f) factorizes the polynomial f into powers of irreducible polynomials. The factorization is actually computed for the canonical associate g of f and the unit u such that $f = ug$ is also returned. For a polynomial f with coefficient ring a field, g is the monic polynomial associated with f while for a polynomial f with coefficient ring $\mathbb{Z}$, g is the associate of f with positive leading coefficient.

Example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> f := 5 * (x^2 + y + z)*(x + y^2 + z)*(x + y + z^2)^2;
> f;
5*x^5 + 5*x^4*y^2 + 10*x^4*y + 10*x^4*z^2 + 5*x^4*z + 10*x^3*y^3 +
    10*x^3*y^2*z^2 + 5*x^3*y^2 + 10*x^3*y*z^2 + 10*x^3*y*z + 5*x^3*y + 5*x^3*z^4
    + 10*x^3*z^3 + 5*x^3*z + 5*x^2*y^4 + 10*x^2*y^3*z^2 + 5*x^2*y^3 +
    5*x^2*y^2*z^4 + 10*x^2*y^2*z + 10*x^2*y^2 + 10*x^2*y*z^3 + 10*x^2*y*z^2 +
    15*x^2*y*z + 5*x^2*z^5 + 10*x^2*z^3 + 5*x^2*z^2 + 10*x*y^4 + 10*x*y^3*z^2 +
    10*x*y^3*z + 5*x*y^3 + 10*x*y^2*z^3 + 10*x*y^2*z^2 + 15*x*y^2*z + 5*x*y*z^4
    + 20*x*y*z^3 + 10*x*y*z^2 + 5*x*z^5 + 10*x*z^4 + 5*y^5 + 10*y^4*z^2 +
    5*y^4*z + 5*y^3*z^4 + 10*y^3*z^3 + 5*y^3*z + 5*y^2*z^5 + 10*y^2*z^3 +
    5*y^2*z^2 + 5*y*z^5 + 10*y*z^4 + 5*z^6
> F, u := Factorization(f);
> F, u;
[
    <x + y + z^2, 2>,
    <x + y^2 + z, 1>,
    <x^2 + y + z, 1>
]
5
```

The function `IsIrreducible`(f) returns true if and only if the polynomial f is irreducible.

Chapter 8

Ideals

In this section, assume that P is a polynomial ring in n variables over the field K.

For an ideal I of the polynomial ring P, a **basis** B of I is an (ideal) generating set of I, i.e., a subset of I such that any element of I can be expressed as a finite sum of products of the form fg where $f \in P$ and $g \in B$. **The Hilbert Basis Theorem states that every ideal I possesses a finite basis**. Within MAGMA , a basis of an ideal I is in fact an ordered sequence of elements of I, possibly with repetitions. Thus a basis of an ideal is different from, say, a vector space basis since it may contain repetitions and zero elements.

The most simple but important problem associated with an ideal is the membership test: given a polynomial $f \in P$ and an ideal I of P, is $f \in I$? To answer this question, a multivariate division algorithm will be developed.

Suppose B is a basis of an ideal I of P and $f \in P$. If f is non-zero, f is said to be **top-reducible** with respect to B if there exists a $g \in B$ such that the **leading monomial** (greatest monomial with respect to the monomial ordering) of g divides the leading monomial of f. If such a case holds, then f can be reduced by g by subtracting the appropriate multiple of g from f which cancels the leading term of f. It is easy to show from the properties of the monomial order that the leading monomial of the new polynomial f_1 (if non-zero) is strictly less than that of f. If the new polynomial f_1 is non-zero yet also top-reducible with respect to B, the same procedure can be applied to it to yield f_2, and so on until an f_k is reached which is either the zero polynomial or is non-zero but is not top-reducible with respect to B. The process does terminate because a monomial order is a well-ordering. The polynomial f_k is called a normal form of f with respect to the basis B. (If f is already zero, then f is a (in fact, the only) normal form of f.)

If a normal form of a polynomial $f \in P$ with respect to the basis B is zero, then it is easy to see that $f \in I$. However, the converse is in general false; i.e., it is possible that a polynomial f is in the ideal I yet a normal form of f with respect to B is non-zero. As an example, let P be the polynomial ring $\mathbb{Q}[x, y, z]$ with lexicographical order and $x > y > z$, let B be the basis $\{x + y, x + z\}$, and let I be the ideal generated by B. Then the polynomial $f = y - z$ is obviously in I yet a normal form of it with respect to B is still f since the leading monomials x and x respectively of the polynomials of B do not divide the leading monomial y of f.

The notion of **Gröbner basis** was introduced to remedy this problem: a basis B of a polynomial ideal I of P is called a Gröbner basis if, for all $f \in P$, $f \in I$ if and only if every normal form of f

with respect to B is zero. The **Buchberger Algorithm** takes an arbitrary basis B of an ideal I and constructs a Gröbner basis for I. This algorithm is of central importance to all computations with ideals of multivariate polynomial rings. A Gröbner basis for an ideal with respect to a fixed monomial order can also be put into a unique form, just like the unique reduced-row echelon form of a matrix. This is always done in MAGMA so that every ideal I possesses a unique Gröbner basis since it has a fixed monomial order.

MAGMA incorporates an efficient implementation of the Buchberger Algorithm which is automatically invoked when necessary. The usual way to create an ideal is by the ideal constructor which is given the polynomial ring P (or an over-ideal) on the left side, and a list on the right side which describes a basis for an ideal. MAGMA remembers the original basis as constructed as the current basis for the ideal until it needs a Gröbner basis: at that point it changes the current basis to the unique Gröbner basis of that ideal and discards the original basis. To force MAGMA to explicitly compute the Gröbner basis of an ideal I, the procedure `Groebner`(I) may be invoked. Note that it is never actually necessary to do this for any ideal computations, MAGMA will automatically compute the Gröbner basis if it needs to do so.

Example:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x + y, x + z>;
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Homogeneous
Basis:
[
    x + y,
    x + z
]
> (y-z) in I;
true
> Groebner(I);
> I;
Ideal of Polynomial ring of rank 3 over Rational Field
Order: Lexicographical
Variables: x, y, z
Homogeneous, Dimension >0
Groebner basis:
[
    x + z,
    y - z
]
```

The following table lists the functions for creating ideals and accessing their bases.

MAGMA	Meaning
`ideal<P|L>`	Ideal of P generated by polynomials in L
`ideal(Q)`	Ideal with user basis Q
`Basis(I)`	Current basis of I
`BasisElement(I,i)`	i-th element of the basis of I
`Coordinates(I,f)`	Coordinates of $f \in I$ with respect to the basis of I
`Groebner(I)`	Procedure explicitly forcing a Gröbner basis to be constructed for I
`GroebnerBasis(I)`	(Unique) Gröbner basis of ideal I
`GroebnerBasis(S)`	(Unique) Gröbner basis of the ideal generated by set or sequence S

Note that it is possible by the `Ideal`(Q) function to create an ideal with a basis such that the basis is remembered by MAGMA so that polynomials of the ideal can be written in terms of this basis by the `Coordinates`(I, f) function (see the *Handbook* for details).

8.1 Ideal Access and Arithmetic

Since ideals of multivariate polynomials are general ideals (and rings) in themselves, many common ideal operations can be performed on them. The following table lists the basic access and arithmetic functions for multivariate polynomial ideals. The table also lists some operations which can be done on polynomials with respect to an ideal.

MAGMA	Meaning
`I+J`	Sum of ideals I and J
`I*J`	Product of ideals I and J
`I^k`	k-th power of ideal I
`ColonIdeal(I,J)`	Colon ideal or ideal quotient $I : J$
`I meet J`	Intersection of ideals I and J
`&meet S`	Intersection of set/sequence S of ideals
`Generic(I)`	Generic polynomial of which I is an ideal
`I eq J`	True iff ideals I and J are equal
`I subset J`	True iff ideal I is contained in ideal J
`IsProper(I)`	True iff ideal I is a proper ideal
`IsZero(I)`	True iff ideal I is the zero ideal
`f in I`	True iff polynomial f is in ideal I
`IsInRadical(f,I)`	True iff polynomial f is in the radical of ideal I
`NormalForm(f,I)`	Normal form of polynomial f with respect to ideal I
`SPolynomial(f,g)`	S-polynomial of polynomials f and g

Example:
An example to demonstrate the ideal arithmetic functions.

```
> P<x, y, z> := PolynomialRing(GF(2), 3);
> I := ideal<P | x^2 + y^2, (y + z)^3 - 1>;
> J := ideal<P | x^2 + y^2, y + z>;
```

```
> M := I meet J;
> M;
Ideal of Polynomial ring of rank 3 over GF(2)
Order: Lexicographical
Variables: x, y, z
Inhomogeneous, Dimension >0
Basis:
[
    y^4 + y + z^4 + z,
    x^2 + y^2
]
> M subset I;
true
>  M subset J;
true
> I * J eq M;
true
> IsInRadical(x + y, I);
true
> (x + y) in I;
false
> (x + y)^2 in I;
true
> ColonIdeal(J, ideal<P|x+y>);
Ideal of Polynomial ring of rank 3 over GF(2)
Order: Lexicographical
Variables: x, y, z
Homogeneous, Dimension >0
Basis:
[
    x + z,
    y + z
]
```

8.2 Quotient Rings

For an ideal I of the polynomial ring $P = K[x_1, ..., x_n]$, it is possible to form the quotient ring $Q = P/I$. This can be done either by the `/` function or by the `quo` constructor. The new indeterminates of the quotient ring can be named by the angle bracket notation as usual.

Example:
The following construction of a quotient ring Q:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> Q<a, b, c> := quo<P | x^3 + y + z, x + y^3 + z, x + y + z^3>;
```

is completely equivalent to the construction:

```
> P<x, y, z> := PolynomialRing(RationalField(), 3);
> I := ideal<P | x^3 + y + z, x + y^3 + z, x + y + z^3>;
> Q<a, b, c> := P / I;
```

The quotient ring Q consists of the residues (normal forms) of the elements of P with respect to the ideal I. Obviously Q has the structure of a vector space over the field K, just like P has.

An extremely important case arises when the dimension of Q as a vector space over K is finite. In such a case, MAGMA provides the function `VectorSpace`(Q) to create a vector space V together with an isomorphism f from Q onto V. This is useful for investigating further structural properties of Q. The following table lists the functions for computing with quotient rings.

MAGMA	Meaning
`CoefficientRing(Q)`	Coefficient ring of quotient ring Q
`Rank(Q)`	Rank of quotient ring Q
`Q.i`	i-th indeterminate of quotient ring Q
`Dimension(Q)`	Dimension of quotient ring as vector space over K
`VectorSpace(Q)`	Vector space isomorphic to Q together with isomorphism
`MatrixAlgebra(Q)`	Matrix algebra isomorphic to Q together with isomorphism

Example:
Continuing the above example.

```
> Dimension(Q);
27
> V, f := VectorSpace(Q);
> [V.i@@f: i in [1..27]];
[
    1,
    c,
    c^2,
    c^3,
    c^4,
    c^5,
    c^6,
    c^7,
    c^8,
    c^9,
    c^10,
    c^11,
    c^12,
    c^13,
    c^14,
    c^15,
    c^16,
    b,
    b*c,
```

```
    b*c^2,
    b*c^3,
    b*c^4,
    b*c^5,
    b*c^6,
    b*c^7,
    b*c^8,
    b^2
]
```

As well as the functions applicable for standard ring elements, the functions `RepresentationMatrix(f)`, `MinimalPolynomial(f)`, `IsUnit(f)`, and `IsNilpotent(f)` may also be applied to elements of finite-dimensional quotient rings (see the *Handbook* for details).

Example:
Suppose the user wishes to find the minimal polynomial of $\theta = \sqrt{2}\sqrt[3]{5}$ over $\mathbb{Q}$. To do this, it suffices to calculate the minimal polynomial of (the residue class of) $x + y$ over $\mathbb{Q}$ in the quotient ring $\mathbb{Q}[x, y]/(x^2 - 2, y^3 - 5)$.

```
> P<x, y> := PolynomialRing(RationalField(), 2);
> Q<a, b> := quo<P | x^2 - 2, y^3 - 5>;
> UP<t> := PolynomialRing(RationalField());
> MinimalPolynomial(a + b);
t^6 - 6*t^4 - 10*t^3 + 12*t^2 - 60*t + 17
```

Chapter 9

Finite Fields

A **finite field** or **Galois field** is a field containing only a finite number of elements q. Such a field exists only if q is a positive power n of a prime number p and, up to isomorphism, there is exactly one field for each such q. The numbers p and n for a field of size p^n are called the characteristic and the degree. A finite field of p^n elements contains a subfield of cardinality p^m when m divides n. In particular, any finite field contains a subfield of prime cardinality p; such a field is called a **prime field**.

In MAGMA it is well possible to work in a finite field created from the specification of p^n only. The first section of this Chapter provides the details. In many situations, however, it is convenient to have more control over the representation of elements of finite fields and of the relations between finite fields. Later sections of this Chapter contain the more advanced mechanism in MAGMA for dealing with that.

9.1 Finite Fields by Cardinality

9.1.1 Constructing Finite Fields

The simplest way of creating the Galois field with q elements in MAGMA is to use the function `FiniteField` (with which `GaloisField` is synonymous, allowing the abbreviation `GF`). The function may be used as `GF`(q) or as `GF`(p, n), with exactly the same result except that in the second version MAGMA does not have the extra task of factorizing q; this may save some time if p is large.

Example:

```
> F32 := GF(2, 5);
> F32;
Finite field of size 2^5
> F32 eq GF(32);
true
```

Given a finite field F with p^n elements, the MAGMA functions `Degree`(F) and `Characteristic`(F) return p and n. The cardinality $q = p^n$ of F is returned by `#`F.

Once a finite field of cardinality p^n has been created, it is possible to define extensions and subfields by use of ext and sub, specifying just the desired degree. For extensions any degree is allowed, for subfields the degree must divide n of course.

Example:

```
> G := GF(11, 3);
> H := ext< G | 4 >;
> I := sub< H | 6 >;
> print I;
Finite field of size 11^6
```

There exist many more sophisticated ways of creating extensions and sub-fields, which we will encounter later.

To define elements in larger fields we use some distinguished element, a generator. Any finite field F in MAGMA is built up as the extension of another finite field G (called the ground field) by an element called the generator of the field. The ground field is either explicit in the construction, for example when we use `ext`, or implicit, such as in the `sub` construction and in the `GF`(p, n) definition. In such implicit cases the ground field is the smallest subfield, i.e., the prime subfield.

Any prime field will be its own ground field, with generator simply 1. There are functions for retrieving the ground field and prime subfield.

Example:

```
> G := GF(11, 3);
> H := ext< G | 4 >;
> PrimeField(G), PrimeField(H);
Finite field of size 11
Finite field of size 11
```

9.1.2 Creation of Elements

The generator of a finite field F has the property that every other element of the field can be written as a polynomial expression in this generator, with coefficients from the ground field G. This polynomial representation is unique if we allow polynomials of degree less than $n - m$ only, where n and m are the degrees of F and G.

The generator can be extracted from the field using `Generator`(G) or `G.1`. but it is much better to assign it to a variable and at the same time determine the way in which elements of the field will be printed by the familiar angle bracket construction on the left-hand side in the definition of the finite field.

Example:

```
> G<g> := GF(11, 3);
> G.1;
g
```

```
> Random(G);
g^29
> g^2-g+10;
g^1158
```

As can be seen here, the elements of this finite field are not printed as a polynomial of degree less than 3 in the generator g. The reason is that there is another representation of elements that is often convenient. The non-zero elements of a finite field form a finite (multiplicative) group that is cyclic of order $q-1$. Once a generator v of this group is found, every element of the field except zero can be written uniquely as v^k with $0 \leq k < q-1$. We will call this the power presentation. An element of the field that generates the multiplicative group is called primitive.

In principle we then have two representations for non-zero elements in any finite field: the polynomial presentation and the power presentation. For small fields (for q less than around 106) both may be used; by default elements are given in the power presentation, but by changing an attribute on the field it is possible to change this.

Example:
Continuing the example above:

```
> AssertAttribute(G, "PowerPrinting", false);
> g^29, g^1158;
g^2 + 9*g + 10
g^2 + 10*g + 10
```

For large fields elements will always be shown in polynomial representation. The reason is that it is not always easy to find a primitive polynomial, and it may be hard to convert between polynomial and power presentations. In general it is not guaranteed that field generators are primitive. The function `IsPrimitive` may be used to check primitivity of an element. Moreover, with `PrimitiveElement`(F) a primitive element can be found.

Example:

```
> H<h> := ext< G | 4>;
> Random(H);
h^827
> IsPrimitive(h);
true
> print IsPrimitive(h^2+h+1);
false
> h^2+h+1;
h^1465
> #MultiplicativeGroup(H);
4095
> Order(h);
4095
> Order(h^2+h+1);
819
```

Notice that the element h is primitive, but $h^2 + h + 1$ is not. Indeed, the order of h is equal to the cardinality of the multiplicative group of H, while the order of $h^2 + h + 1$ is not.

9.2 Extensions of Finite Fields

9.2.1 Extensions by a Given Defining Polynomial

Every Galois field E of size p^n, where $n > 1$, is a simple algebraic extension of some field F of size p^m, where $m < n$ and m divides n. F is called the ground field of E, and is isomorphic to a subfield of E. If F is not a prime field, that is, if $m > 1$, then it is in turn an extension of a smaller field with the same characteristic p. This chain will eventually terminate at `GF`(p), the prime field of E. The integer $d = \frac{n}{m}$ is the degree of the extension field E relative to the ground field F, whereas n is the absolute degree of E, that is, its degree relative to the prime field.

The field E is created as a simple algebraic extension of degree d of F. In other words, a root of an irreducible polynomial f of degree d with coefficients in F is adjoined to F in order to create E. (For any given finite field and degree, there always exists at least one irreducible polynomial of degree d over the field.) In MAGMA , the `ext` constructor is used to create an extension. One form of this constructor is

`ext<` F | f `>`

If F is a finite field and f is an irreducible polynomial f of degree d with coefficients in F, this creates an extension of degree d of the ground field F, with f as its defining polynomial.

Example:
Suppose that the user wishes to construct a degree-4 extension of $\mathbb{F}_9$ using the defining polynomial $f = x^4 + g^6x^2 + g^5$ over $\mathbb{F}_9$, where g is a primitive element of the field.

```
> gf9<g> := GF(9);
> Pgf9<x> := PolynomialRing(gf9);
> f := x^4 + g^6*x^2 + g^5;
```

A check should be made that f is irreducible, and then the extension may be defined.

```
> IsIrreducible(f);
true
> gf9e4<alpha> := ext< gf9 | f >;
> gf9e4;
Finite field of size 3^8
```

The new extension, `gf9e4`, has `alpha` as its generator. This generator is a root of f, the defining polynomial, so if it is substituted into f, the result is zero:

```
> Evaluate(f, alpha);
0
```

In fact, although f has no roots over its own coefficient field, the corresponding polynomial with coefficients in the extension field has $d = 4$ roots, including `alpha`.

```
> Roots(f);
[]
> Roots(f,gf9e4);
[ <alpha^3 + alpha, 1>, <alpha, 1>, <2*alpha^3 + 2*alpha, 1>, <2*alpha, 1> ]
```

The functions `PrimeField`, `GroundField` and `Degree` may be used to gain information about a field.

Example:
Continuing the example:

```
> print PrimeField(gf9e4);
Finite field of size 3
> print GroundField(gf9e4);
Finite field of size 3^2
> print Degree(gf9e4);
8
> print Degree(gf9e4, gf9);
4
```

Note that `Degree` has two versions: `Degree`(E) is the absolute degree n over the prime field of a field E with p^n elements, and `Degree`(E, F) is the degree $d = \frac{n}{m}$ of E considered as an extension of a field F with p^m elements.

It is possible to avoid explicitly creating the polynomial ring for the defining polynomial by using the constructor `ExtensionField`, a variant of the `ext` constructor. It has the form

`ExtensionField<` F `,` indeterminate | f `>`

where the polynomial f is expressed in terms of the indeterminate.

Example:
For instance, the following statement demonstrates how to create `gf9e4` without first creating `Pgf9` and its indeterminate `x`.

```
> gf9e4<alpha>:=ExtensionField< gf9, x | x^4+g^6*x^2+g^5 >;
```

The indeterminate (x in this example) does not remain defined as a identifier or printname. It is only used for stating the defining polynomial.

9.2.2 Extensions by a Given Degree

Another way of creating an extension E of a finite field F is to specify the required degree d instead of the defining polynomial in the `ext` constructor:

`ext<` F | degree d of extension over ground field F `>`

When the `ext` constructor is used in this fashion, MAGMA selects a suitable defining polynomial for the extension. The irreducible polynomial chosen, which is available from the function `DefiningPolynomial`(E), is the same as the polynomial returned by `IrreduciblePolynomial`(F, d).

Example:
For instance, `gf9e4e3` may be constructed as a degree-3 extension of `gf9e4`, without specifying a defining polynomial.

```
> gf9e4e3<beta> := ext< gf9e4 | 3 >;
> gf9e4e3;
Finite field of size 3^24
> defgf9e4e3 := DefiningPolynomial(gf9e4e3);
> defgf9e4e3;
$.1^3 + (g^7*alpha^3 + g^6*alpha^2 + g^3*alpha)*$.1 + g^3*alpha^3 + 2*alpha^2
+ g^6*alpha
> defgf9e4e3 eq IrreduciblePolynomial(gf9e4, 3);
true
```

Notice from the output form of `defgf9e4e3` that there is no printname for the indeterminate in the defining polynomial, so the special symbol `$.1` is employed. If the user wishes to give the indeterminate a printname and assign it to an identifier, there are two methods: either the parent polynomial ring may be defined, together with its indeterminate, or (more simply) angle brackets may be used when assigning `defgf9e4e3`:

```
> defgf9e4e3<y> := DefiningPolynomial(gf9e4e3);
> defgf9e4e3;
y^3 + (g^7*alpha^3 + g^6*alpha^2 + g^3*alpha)*y + g^3*alpha^3 + 2*alpha^2
+ g^6*alpha
```

Both forms of `ext`, as well as `ExtensionField`, optionally return a second value, which is the embedding monomorphism $\varphi : F \to E$. This map has the same effect as coercion of an element of F into E using the `!` operator.

9.2.3 Splitting Fields

The defining polynomial for an extension formed using the ext constructor has to be irreducible. However, it is also possible to form an extension using a non-irreducible polynomial. Given any polynomial p over F, `SplittingField`(p) constructs the smallest-degree extension E of F such that the polynomial p factors completely into linear factors when considered as a polynomial over E.

Example:
Suppose p is the polynomial defined below. The coefficient field of p is `gf9`. Over this field, p has two non-linear irreducible factors, but no roots.

```
> p := MinimalPolynomial(alpha^3+alpha, gf9) * MinimalPolynomial(g*beta^2, gf9);
> Factorization(p);
[
    <x^4 + g^6*x^2 + g^5, 1>,
    <x^12 + g*x^11 + g*x^10 + 2*x^9 + 2*x^8 + g^3*x^7 + g^5*x^6 + g*x^5
    + g^3*x^4 + 2*x^3 + g^3*x^2 + g^6*x + 2, 1>
]
> Roots(p);
```

```
[]
```

Over the field `gf9e4`, which contains `alpha^3+alpha`, p has the 4 roots corresponding to the degree-4 irreducible factor shown above.

```
> Roots(p,gf9e4);
[ <alpha^3 + alpha, 1>, <alpha, 1>, <2*alpha^3 + 2*alpha, 1>, <2*alpha, 1> ]
```

However, over the splitting field of p, the degree-16 polynomial p has 16 roots.

```
> Spl := SplittingField(p);
> Spl;
Finite field of size 3^24
> #Roots(p,gf9e4e3);
16
> <alpha^3+alpha,1> in Roots(p,gf9e4e3);
true
> <g*beta^2,1> in Roots(p,gf9e4e3);
true
```

In this collection of roots, the 4 roots found previously appear once again. As expected from the construction of the polynomial, `alpha^3+alpha` and `g*beta^2` are among the roots.

Given a set T of polynomials over a finite field F, `SplittingField`(T) creates the smallest-degree extension field E such that every polynomial in T factors completely into linear factors over E.

9.2.4 Roots of Unity in Extension Fields

Given an $n \in \mathbb{Z}$ and a finite field K, the function `RootOfUnity`(n, K) returns a primitive nth root of unity in the smallest possible extension field of K.

Example:

```
> gf9<g> := GF(9);
> r := RootOfUnity(23, gf9);
> L<e> := Parent(r);
> L;
Finite field of size 3^22
> r;
g^3*e^10 + g^5*e^9 + g*e^8 + g^3*e^7 + g^3*e^6 + g^5*e^5 +
2*e^4 + e^3 + g^2*e^2 + g^2*e + g^3
> Order(r);
23
```

9.3 Subfields of Finite Fields

The operation approximately opposite to the creation of an extension is the creation of a subfield. The `sub` constructor is used for this purpose, but in the case of finite fields, it has two versions (like the `ext` constructor):

`sub<` field | a `>`

where a is an element of the field, and

`sub<` field | m `>`

where m is a positive integer dividing the absolute degree of the original field. The first of these returns the subfield generated by a, and the second returns a subfield of absolute degree m. Both forms of `sub` optionally return the embedding monomorphism from the subfield to the main field as a second value. This map has the same effect as a standard coercion.

Example:
The following lines create two fields of size 3^4, as subfields of `gf9e4e3`.

```
> gf9e4e3s4A<gammaA> := sub< gf9e4e3 | alpha^2 >;
> gf9e4e3s4B<gammaB> := sub< gf9e4e3 | 4 >;
> gf9e4e3s4A;
Finite field of size 3^4
> gf9e4e3s4B;
Finite field of size 3^4
```

Although these fields are isomorphic, because they have the same size, they happen to have different defining polynomials, and their generators are not equal. The coercion mapping gives the isomorphism between the fields.

```
> DefiningPolynomial(gf9e4e3s4A);
$.1^4 + 2*$.1 + 2
> DefiningPolynomial(gf9e4e3s4B);
$.1^4 + 2*$.1^3 + 2
> gammaA eq gammaB;
false
> gf9e4e3s4A ! gammaB;
gammaA^31
> gf9e4e3s4B ! gammaA;
gammaB^31
```

With respect to the field `gf9e4`, the generator of `gf9e4e3s4A` is `alpha2`, as would be expected, but the generator of `gf9e4e3s4B` is `g7alpha2`:

```
> print gf9e4 ! gammaA;
alpha^2
> print gf9e4 ! gammaB;
g^7*alpha^2
```

9.4 Associated Polynomials

Several polynomials are associated with finite fields, including defining polynomials, irreducible polynomials as explained above. The following table gives a list of the polynomial functions for finite fields.

MAGMA	Meaning
`DefiningPolynomial(F)`	Polynomial over the ground field of F, defining F as an extension of the ground field
`DefiningPolynomial(F,S)`	Polynomial over subfield S of F, defining F as an extension of S
`IrreduciblePolynomial(F,m)`	Irreducible polynomial of degree m over F
`AllIrreduciblePolynomials(F,m)`	Set of the monic irreducible polynomials of degree m over F
`IsIrreducible(f)`	**true** if univariate polynomial f over F is irreducible
`PrimitivePolynomial(F,m)`	A primitive polynomial of degree m over F (i.e., irreducible, and having a primitive root of the degree-m extension field of F as a root)
`IsPrimitive(f)`	**true** if univariate polynomial f over F is primitive
`MinimalPolynomial(a)`	Minimal polynomial of $a \in F$ over ground field of F
`MinimalPolynomial(a,S)`	Minimal polynomial of $a \in F$ over subfield S of F
`CharacteristicPolynomial(a)`	Characteristic polynomial of $a \in F$ over ground field of F
`CharacteristicPolynomial(a,S)`	Characteristic polynomial of $a \in F$ over subfield S of F

A few of these functions resemble the `Degree` function in that they can take one argument or two arguments. If only one argument is given, the operation is performed with respect to the ground field, whereas if two arguments are given, the operation is performed with respect to the specified subfield. (However, if no subfield is stated in the `Degree` function, it is understood to be the prime field, not the ground field.) For instance, the function `MinimalPolynomial` returns the minimal polynomial of a given element a of the finite field F. If a subfield S of F is given as the second argument, the value returned is the minimal polynomial over the coefficient field S. Otherwise, it is the minimal polynomial over the ground field of F.

9.5 Operations on Finite Field Elements

MAGMA has functions for several standard operations on elements of finite fields. Some of them are listed in the table below. Notice that for those functions that take an optional subfield as a second argument, whenever the subfield is omitted the default subfield is the prime field, not the ground field as in the polynomial functions explained above.

MAGMA	Meaning
`Norm(a)`	Absolute norm of a (norm relative to prime field)
`Norm(a,S)`	Norm of a relative to subfield S
`Trace(a)`	Absolute trace of a (trace relative to prime field)
`Trace(a,S)`	Trace of a relative to subfield S
`Log(a)`	Discrete logarithm of a ($\neq 0$), with 'base' the defining primitive element w, i.e., n such that $w^n = a$
`Order(a)`	Multiplicative order of a ($\neq 0$)
`FactoredOrder(a)`	Multiplicative order of a ($\neq 0$) in factored form (as a factorization sequence)
`Sqrt(a)`	A square root of a, if an answer exists
`IsSquare(a)`	If a is square, returns **true** and a square root, else returns **false**
`Root(a,n)`	An n-th root of a, if an answer exists
`Eltseq(a)`	Length-d sequence containing the coefficients of $a \in F$ as a polynomial over S in the generator of F over S, where S is the ground field of F and d is the degree of F over S
`Eltseq(a,S)`	Length-d sequence containing the coefficients of $a \in F$ as a polynomial over S in the generator of F over S, where S is a subfield of F and d is the degree of F over S
`Eltseq(a)`	Sequence as for `Eltseq`(a, S), where S is the ground field of F
`Seqelt(Q,F)`	Given a sequence Q as described for `Eltseq`, returns the corresponding element of F

One of the most important functions for finite field elements is `Log`(a). It uses the primitive element $w \in F$ that Magma returns from the function `PrimitiveElement`(F). By definition of w, every non-zero $a \in F$ may be expressed as w^n for a unique $n \in \{0, \ldots, \#F-2\}$. `Log`$(a)$ returns this value n.

Example:

```
> pr := PrimitiveElement(gf9e4);
> pr;
g^6*alpha^3 + g^2*alpha^2 + g^7*alpha + 2
> Log(alpha);
1517
> pr^1517;
alpha
> Log(gf9e4!g);
820
> pr^820;
g
```

The `Log` function only applies to moderately small fields, for which MAGMA can calculate the complete table of logarithms.

`ElementToSequence`(a, S) and `SequenceToElement`(Q, S), which are commonly abbreviated to `Eltseq`(a, S) and `Seqelt`(Q, F), convert between an element a of a finite field F and its representation as a polynomial in the generator of F over a subfield S, with coefficients in S. `Eltseq`

returns the coefficients of the polynomial in a length-d sequence Q, where d is the degree of F over S, and `Seqelt` returns the field element a. If `Eltseq` is used in the form `Eltseq`(a), then the subfield S is understood to be the ground field of F.

Example:

```
> Eltseq(pr);
[ 2, g^7, g^2, g^6 ]
> Eltseq(pr, GF(3));
[ 0, 0, 1, 1, 2, 0, 2, 2 ]
> Seqelt([ 2*alpha, 1, g*alpha], gf9e4e3);
g*alpha*beta^2 + beta + 2*alpha
```

9.6 Finite Fields as Vector Spaces

If E is a degree-d extension of F, then there is a natural isomorphism between E and the d-dimensional vector space $E^{(d)}$. Every element of E may be written uniquely as a linear combination of the basis elements $1, \omega, \omega^2, \ldots, \omega^{d-1}$, where ω is the generator of E over F and the scalars are elements of F. The function `VectorSpace`(E, F) returns this vector space, and the corresponding isomorphism from E to the vector space.

Example:

```
> V, phi := VectorSpace(gf9e4, gf9);
> V, phi;
Full Vector space of degree 4 over GF(3^2)
Mapping from: FldFin: gf9e4 to ModTupFld: V
```

The mapping `phi` converts a given element of `gf9e4` to a vector of |V|:

```
> (g^5*alpha^3 + g^5*alpha^2 + g^3*alpha + 2 ) @ phi;
(  2 g^3 g^5 g^5)
```

The correspondence between the element and its vector space representation is obvious.

Instead of using the powers of the generator as a basis, the user may choose a basis for the vector space. If B is a suitable sequence of elements of F, then `VectorSpace`(E, F, B) returns the vector space and isomorphism for that basis.

For example, a normal basis for E over F is a basis of the form $a, a^q, \ldots, a^{q^{d-1}}$. Such a basis has the property that exponentiation of an element by a power of q can be achieved by rotating the vector representing that element. If $a \in E$ forms a normal basis, it is called a normal element.

The function `IsNormal`(a) tests whether a forms a normal basis over the prime field of E, and `NormalElement`(E) returns a normal element for the field E considered over its prime field. (For calculations relative to an arbitrary subfield S, the functions `IsNormal`(a, S) and `NormalElement`(E, S) should be used instead.)

Example:

```
> gf3e5:=GF(3,5);
> a:=NormalElement(gf3e5);
> B:=[a^(3^i): i in [0..4]];
> V,f:=VectorSpace(gf3e5,GF(3),B);
> V,f;
Full Vector space of degree 5 over GF(3)
Mapping from: FldFin: gf3e5 to ModTupFld: V
> w:=Random(gf3e5);
> w @ f;
(0 0 1 2 0)
> (w^3) @ f;
(0 0 0 1 2)
> (w^27) @ f;
(1 2 0 0 0)
```

9.7 Finite Fields as Matrix Algebras

Suppose E is a degree-d extension of a finite field F. (F does not have to be the ground field of E; it may be isomorphic to any subfield of E.) As was shown above, E is isomorphic to the d-dimensional vector space $E^{(d)}$. However, a stronger mathematical statement can be made. E is also isomorphic to a matrix algebra of degree d and dimension d over F. The advantage of this representation is that field elements as matrices can be multiplied and inverses can be found, whereas field elements as vectors can only be added and made negative. Of course, elements of E which are in the ground field F correspond to multiples of the identity matrix, so they still have a scalar action. The function `MatrixAlgebra`(E, F) returns this matrix algebra A, and the corresponding isomorphism from E to A, such that the generator of E maps to the generator of A.

Example:

```
> M<G>, isoM := MatrixAlgebra(gf9e4, gf9);
> M, isoM;
Matrix Algebra of degree 4 and dimension 4 with 1 generator over GF(3^2)
Mapping from: FldFin: gf9e4 to AlgMat: M
> G eq isoM(alpha);
true
> isoM(alpha^2+1) eq G^2+1;
true
```

Observe that `G`, the generator of `M`, is isomorphic to `alpha`, the generator of `gf9e4`. The isomorphism is `isoM`.

The generator of the matrix algebra is constructed as the companion matrix of the field's defining polynomial. It has a line of 1s above the main diagonal, and 0s elsewhere, except for the last row.

Example:

```
> G;
[  0   1   0   0]
[  0   0   1   0]
[  0   0   0   1]
[  g   0 g^2   0]
> defgf9e4 := DefiningPolynomial(gf9e4);
> defgf9e4;
x^4 + g^6*x^2 + g^5
> G eq CompanionMatrix(defgf9e4);
true
```

In general, the top row of the matrix indicates the vectorial representation of the element as a sum of powers of the generator up to $d-1$.

Example:

```
> m := 2 + G^2 + G^3;
> m;
[  2   0   1   1]
[  g   2 g^2   1]
[  g   g   g g^2]
[g^3   g g^7   g]
```

Functions such as `MinimalPolynomial`, `CharacteristicPolynomial` and |Order| can be applied to elements of the matrix algebra just as for field elements.

Example:

```
> print Order(m);
656
> print Order(2 + alpha^2 + alpha^3);
656
```

Chapter 10

Vector Spaces and Matrix Spaces

The most general kind of module which can be constructed in MAGMA is a module of dimension n over a ring R. Magma offers two ways of viewing such structures: as ($R-$)tuple spaces with an embedded basis, or as ($R-$)modules with a reduced basis. This chapter discusses the former view only.

If the ring over which a tuple space is defined is a field K, then the structure which emerges is a K-space, more commonly known as a (*finitely-generated*) vector space. Its elements are row vectors. The category of such structures in MAGMA is `ModTupFld`. (The field K must be defined as a member of a `Fld` category, so that MAGMA knows it is a field.) For the sake of clarity and familiarity, this chapter concentrates on vector spaces, although most of the operations are also available for R-spaces, that is, tuple spaces over a general ring R (category `ModTupRng`).

A further topic of this chapter is matrix spaces. Again, the emphasis will be on K-matrix spaces (`ModMatFld`), but the operations for R-matrix spaces in general (`ModMatRng`) closely resemble them. The elements of matrix spaces are represented as rectangular matrices, and may be considered either as vector-like objects printed as matrices, or as homomorphisms from one vector space to another.

10.1 Constructing the Full Vector Space

Every vector space of finite dimension n over a field K is isomorphic to the vector space K^n, that is, the space of n-dimensional coordinates whose entries are elements of K. This space is known as the full vector space. It may be created in MAGMA using the function `VectorSpace`(K, n), or `KSpace`(K, n).

Example:
The full 5-dimensional vector space over the rational field $\mathbb{Q}$ may be constructed as follows.

```
> Q := RationalField();
> V5 := VectorSpace(Q, 5);
```

It is not necessary to assign the field to an identifier before creating the vector space. Thus the following line will create the 3-dimensional vector space `V3F11` over the finite field with eleven elements.

```
> V3F11 := VectorSpace(GF(11), 3);
```

The function `Basis`(V) returns the basis of a vector space V. If V is a full vector space, then it returns the standard basis, in which the i-th basis vector is the vector that has the identity of the field K as its i-th entry, and the zero of K in all other coordinate positions.

Example:

```
> V3F11;
Full Vector space of degree 3 over GF(11)
> Basis(V3F11);
[
    ( 1  0  0),
    ( 0  1  0),
    ( 0  0  1)
]
```

For full vector spaces created using `VectorSpace`(K), the generators are the same as the basis vectors, and the i-th generator, `V.i`, is the i-th basis vector.

Example:

```
> Generators(V3F11);
{
    ( 0  0  1),
    ( 1  0  0),
    ( 0  1  0)
}
> V3F11.2;
( 0  1  0)
```

10.2 Constructing the Full Matrix Space

Computations with $m \times n$ matrices in MAGMA may occur in several ways. Firstly, such matrices are elements of a matrix space, most obviously the full matrix space containing all $m \times n$ matrices over the field K. The function `KMatrixSpace`(K, m, n) returns this space.

Example:
The following line assigns the matrix space of 2×3 matrices over the rationals to the identifier `M23`.

```
> M23 := KMatrixSpace(Q, 2, 3);
```

The standard basis of a full matrix space consists of all the 2×3 matrices with a single 1 entry and zeros elsewhere. Over the basis sequence, this 1 moves along each position within each row, starting with all of the first row, followed by the second row, and so on. This order is called *row-major* order.

```
> Basis(M23);
```

```
[
    [1 0 0]
    [0 0 0],

    [0 1 0]
    [0 0 0],

    [0 0 1]
    [0 0 0],

    [0 0 0]
    [1 0 0],

    [0 0 0]
    [0 1 0],

    [0 0 0]
    [0 0 1]
]
```

Matrix spaces containing $m \times n$ matrices can also be regarded by MAGMA as vector spaces $K^{m \times n}$. Therefore all the vector operations can be used for matrices too. These operations will be explained in this chapter.

Thirdly, MAGMA can view $m \times n$ matrices as linear transformations or homomorphisms between two vector spaces K^m and K^n. Therefore the matrix space $K^{m \times n}$ can also be created by means of the homomorphism module function `Hom`(M, N).

Example:
Continuing our example, `M23` may be compared with the homomorphism module consisting of the homomorphisms from the vector space $\mathbb{Q}^2$ to the vector space $\mathbb{Q}^3$.

```
> V2 := VectorSpace(Q, 2);
> V3 := VectorSpace(Q, 3);
> H23 := Hom(V2, V3);
> H23 eq M23;
true
```

10.3 Vectors and Matrices

10.3.1 Creating Vectors and Matrices

A vector may be created in MAGMA by coercing the sequence of its components into its parent vector space, using a `!` symbol.

Example:
For instance, the vector $(-2, \frac{3}{4}, 12, 0, -\frac{8}{5})$ in $\mathbb{Q}^5$ may be assigned to the identifier v as follows.

```
> V5 := VectorSpace(Q, 5);
> v:=V5![-2, 3/4, 12, 0, -8/5];
```

Vectors whose components follow some mathematical pattern can be ex- pressed easily by exploiting the sequence constructors.

Example:

```
> w := V5![3/4*(-1)^i: i in [1..5]];
> w;
(-3/4  3/4 -3/4  3/4 -3/4)
```

The zero vector of a vector space V is `V!0`. The `IsZero`(v) function returns true when its argument is the zero vector.

Matrices are constructed in the same way, by regarding the matrix space as a vector space of dimension mn. The entries must be listed in row-major order, the same order as for the standard basis. (This is the same method for matrix creation as is used in matrix rings and matrix groups.)

Example:
The following lines construct two 3×7 matrices with real coefficients.

```
> R2 := RealField(2);
> M37 := KMatrixSpace(R2, 3, 7);
> m1:=M37![0,0,0,4.5,1,1,-6.7,0,0,0,0,0,0,0,1,1,0,0,0,1,0];
> m1;
[0.00 0.00 0.00 4.5 1.0 1.0 -6.7]
[0.00 0.00 0.00 0.00 0.00 0.00 0.00]
[1.0 1.0 0.00 0.00 0.00 1.0 0.00]
> m2 := M37![ (21 - t) / 5 : t in [0..20] ];
> m2;
[4.2 4.0 3.8 3.6 3.4 3.2 3.0]
[2.8 2.6 2.4 2.2 2.0 1.8 1.6]
[1.4 1.2 1.0 0.80 0.60 0.40 0.20]
```

Given a vector or matrix u, the function `Eltseq`(u) returns the sequence of its components.

Example:

```
> Eltseq(w);
[ -3/4, 3/4, -3/4, 3/4, -3/4 ]
> Eltseq(m2);
[ 4.2, 4.0, 3.8, 3.6, 3.4, 3.2, 3.0, 2.8, 2.6, 2.4, 2.2, 2.0, 1.8, 1.6, 1.4,
1.2, 1.0, 0.80, 0.60, 0.40, 0.20 ]
```

If U is a vector space or matrix space defined over a finite field, then `Random`(U) returns a random element of U. (U has to be finite because otherwise MAGMA has no way of constructing random elements.)

Example:

```
> for n in [1..4] do
for> Random(V3F11);
for> end for;
( 7 10  8)
( 9  1  3)
( 0  5  4)
( 3  5  6)
```

10.3.2 Arithmetic and Functions

The operators and functions in this subsection apply to both matrices and vectors. If the operations are performed on matrices, MAGMA considers the matrices to be vectors, by treating them in row-major order, but the output for matrices remains in rectangular form.

Arithmetic on MAGMA vectors and matrices is performed with the usual operators. Addition and subtraction are performed component-by-component, so the vectors or matrices in the operation must be of the same size. Scalar multiplication is also available.

Example:
The vector $y = v - \frac{2}{5}w$ can be assigned as follows.

```
> y := v - 2/5*w;
> y;
(-17/10   9/20 123/10  -3/10 -13/10)
```

Normalization is a particular type of automatic scalar multiplication. Given a vector or matrix u, the function `Normalize`(u) returns u divided by its first non-zero component.

Example:

```
> Normalize(y);
(      1   -9/34 -123/17    3/17   13/17)
```

The normalized form of a zero vector is the same as the vector itself.

Given u_1 and u_2 in the same space, `InnerProduct`(u, v) returns their inner product according to whatever inner product is defined on the space (usually the Euclidean norm).

Example:
The following line calculates the inner product of vectors v and w in `V5` with respect to the Euclidean norm, so the output is the sum of the products of corresponding components of v and w.

```
> InnerProduct(v, w);
-459/80
```

10.3.3 Indexing Vectors and Matrices

The usual `v[i]` notation may be used to inspect or modify the i-th component of a vector v.

Example:

```
> y[2];
9/20
> y[2] := 1000;
> y;
(-17/10   1000 123/10  -3/10 -13/10)
```

The indexing of matrices is slightly different. To obtain an individual entry of a matrix, two coordinates are required. Given a matrix M, the entry in the i-th row and j-th column is `M[i,j]`.

Example:

```
> m2[2, 6];
1.8
```

By contrast, `M[i]` means the vector which is the i-th row of M.

Example:

```
> m2[3];
(1.4 1.2 1.0 0.80 0.60 0.40 0.20)
> IsZero(m1[2]);
true
```

There is no way of obtaining a column of M directly, since MAGMA's vectors are row vectors. The best method is to transpose M, using the function `Transpose`(M), and then extract a row from the transposed form.

Example:
The following lines show how to obtain the fourth column of `m2` as a row vector.

```
> m2tr := Transpose(m2);
> m2tr[4];
(3.6 2.2 0.80)
```

The function `Support`(u) returns the set containing the indexes of all the non-zero entries of a vector or matrix u. If u is a matrix, then the indexes are given as tuple pairs $< i, j >$.

Example:

```
> Support(v);
{ 1, 2, 3, 5 }
> Support(m1);
{ <1, 4>, <3, 6>, <3, 2>, <1, 7>, <1, 6>, <1, 5>, <3, 1> }
```

10.3.4 Blocks within Matrices

Given an element A of a matrix space over K, any $p \times q$ block within it may be considered as a matrix (in particular, an element of the space of $p \times q$ matrices over K). The following table lists the operations connected with blocks within matrices, and some examples are given below.

MAGMA	Meaning
`Submatrix(A, i, j, p, q)`	$p \times q$ submatrix of A, starting at row i and column j
`InsertBlock(~A, R, i, j)`	Given $p \times q$ matrix R, changes matrix A by replacing the $p \times q$ submatrix starting at row i and column j with R
`HorizontalJoin(A, B)`	Matrix formed by joining matrices A and B horizontally, where A and B have the same number of rows
`HorizontalJoin(Q)`	Matrix formed by joining sequence Q of matrices horizontally, where the matrices all have the same number of rows
`VerticalJoin(A, B)`	Matrix formed by joining matrices A and B vertically, where A and B have the same number of columns
`VerticalJoin(Q)`	Matrix formed by joining sequence Q of matrices vertically, where the matrices all have the same number of columns
`DiagonalJoin(A, B)`	Matrix formed by joining matrices A and B along main diagonal, with zeros elsewhere
`DiagonalJoin(Q)`	Matrix formed by joining sequence Q of matrices along main diagonal, with zeros elsewhere

10.3.5 Subspaces and Quotient Spaces

Vector subspaces and quotient spaces may be formed in MAGMA using the customary sub and quo constructors.

Example:
For instance, the following statement assigns to `V5s` the subspace of `V5` generated by w, the third generator of `V5` and the vector $(-5, 5, 7, 5, 5)$.

```
> V5s := sub< V5 | w, V5.3, [-5, 5, 7, 5, -5] >;
```

The information listed to the right of the | symbol specifies the generators for the subspace. In this example they are given explicitly as vectors. Other possible objects on the right of the | symbol are subspaces of the main space, and sets or sequences containing these; in this case, the vectors used to generate the subspace will be the generating elements of these subspaces.

Not all vector spaces are represented internally using the generators provided by the user. For the sake of efficiency, MAGMA employs an echelonized basis for this purpose, and the vectors of this basis may differ from those vectors given by the user. However, the generators are all stored (even if they are not linearly independent), and are available in the user-specified order (under the names `V5s.1`, `V5s.2` and `V5s.3` for the example above).

Example:

```
> V5s;
Vector space of degree 5, dimension 2 over Rational Field
Generators:
(-3/4  3/4 -3/4  3/4 -3/4)
(   0    0    1    0    0)
(  -5    5    7    5   -5)
Echelonized basis:
( 1 -1  0 -1  1)
( 0  0  1  0  0)
>
> 10*V5s.2 + V5s.3;
(-5  5 17  5 -5)
```

The quotient constructor `quo` creates the quotient of the main space by the subspace described by whatever is on the right of the | symbol.

Example:
The following lines demonstrate two ways to create the quotient `V5q` of `V5` by `V5s`.

```
> V5q := quo< V5 | V5s >;
```

or alternatively

```
> V5q := quo< V5 | w, V5.3, [-5, 5, 7, 5, -5] >;
> V5q;
Full Vector space of degree 3 over Rational Field
```

The `sub` and `quo` constructors can return mappings as their second return value. These mappings are the inclusion homomorphism and the natural homomorphism, respectively.

Example:
For instance, we can construct the natural homomorphism `phi` which maps `V5` to the quotient space `V5q`. In accordance with vector space theory, the complement `C` of `V5s` is isomorphic to `V5q`, since `C` and `V5q` have the same dimension.

```
> V5q, phi := quo< V5 | w, V5.3, [-5, 5, 7, 5, -5] >;
> phi;
Mapping from: ModTupFld: V5 to ModTupFld: V3
> C := Complement(V5, V5s);
> C;
Vector space of degree 5, dimension 3 over Rational Field
Echelonized basis:
(0 1 0 0 0)
(0 0 0 1 0)
(0 0 0 0 1)
> C @ phi;
Full Vector space of degree 3 over Rational Field
```

```
> C @ phi eq V5q;
true
```

The following line verifies that the kernel of `phi` is `V5s`.

```
> print Kernel(phi) eq V5s;
true
```

If the user has constructed a subspace or quotient space U of a vector space V, without obtaining the corresponding mapping, the mapping can be obtained at some later stage from the function `Morphism`(U, V). It returns the homomorphism as a matrix.

10.3.6 Operations on Vector Spaces

The following lists several of the MAGMA operations on vector spaces. Note carefully the difference between generators and basis vectors.

MAGMA	Meaning
`CoefficientField(V)`, `BaseField(V)`	Field K of K-vector space V
`Dimension(V)`	Dimension of V
`Generic(V)`	Full vector space K^n of which V is a subspace
`OverDimension(V)`, `Degree(V)`	Given subspace V of K^n, returns n
`OverDimension(v)`, `Degree(v)`	Given element v of subspace of K^n, returns n
`Generators(V)`	Set of the generators of V, as given to MAGMA (e.g., in `sub`)
`Ngens(V)`	Number of generators of V
`V.i`	i-th generator of V
`Basis(V)`	Sequence of basis vectors of V, as calculated by MAGMA
`BasisMatrix(V)`	Matrix whose rows are the basis vectors of V
`BasisElement(V,i)`	i-th basis vector of V
`U1 + U2`	Sum of subspaces $U1$ and $U2$
`U1 meet U2`	Intersection of subspaces $U1$ and $U2$
`Complement(V,U)`	Complement for subspace U of V, as a subspace of V

10.3.7 Operations with Linear Transformations

Given any two K-vector spaces V and W of over-dimension (degree) m and n, the set of linear transformations from V to W is the homomorphism module `Hom`(V, W). Every linear transformation $V \to W$ can be written as a matrix A of size $m \times n$ with entries in K. The effect of A on V may be investigated using mapping operations.

Example:
Let g be a primitive element for $\mathbb{F}_9$, and let V and W be certain subspaces of the vector spaces $\mathbb{F}_9^4$ and $\mathbb{F}_9^6$.

```
> gf9<g>:=GF(9);
> V:=sub<VectorSpace(gf9,4)|[ g, 1, g^7, g ],[ g^3, 0, 0, 1 ]>;
```

```
> V;
Vector space of degree 4, dimension 2 over GF(3^2)
Generators:
(  g   1 g^7   g)
(g^3   0   0   1)
Echelonized basis:
(  1   0   0 g^5)
(  0   1 g^7 g^3)
> W:=sub<VectorSpace(gf9,6)|[ 2, 2, 0, g, g, 0 ],[ g^3, g, 2, g^6, 1, g^6 ],
[ 0, g^3, g^7, g^5, 0, g^5 ]>;
> W;
Vector space of degree 6, dimension 3 over GF(3^2)
Generators:
(  2   2   0   g   g   0)
(g^3   g   2 g^6   1 g^6)
(  0 g^3 g^7 g^5   0 g^5)
Echelonized basis:
(  1   0   0   1 g^5 g^3)
(  0   1   0 g^6   0 g^7)
(  0   0   1 g^2   0   1)
> H:=Hom(V,W);
> H;
KMatrixSpace of 4 by 6 matrices and dimension 18 over GF(3^2)
```

Let A equal the sum of the second generator of H and twice the thirteenth generator of H. It has domain V and codomain W.

```
> A:=H.2 + 2*H.13;
> A;
[  0   1   0   0   0   0]
[  0   0   0   0   0   0]
[  2   0   0 g^7   g g^5]
[  0   0   0   g   0 g^2]
> Domain(A) eq V and Codomain(A) eq W;
true
```

The product of an element of V and a is an element of W.

```
> v:=Random(V);
> w:=v*A;
> w;
(g^6 g^5   0   g g^3 g^7)
> w in W;
true
```

It is also possible to multiply a subspace of V by A, so as to obtain a subspace of W. The image of a is returned by the function `Image`(A). It is equal to the product of the whole domain by A.

Example:

```
> I := Image(A);
> I;
Vector space of degree 6, dimension 2 over GF(3^2)
Echelonized basis:
(  1   0   0   1 g^5 g^3)
(  0   1   0 g^6   0 g^7)
> I eq V*A;
true
```

The image of A should be distinguished from the row space of A. The function `RowSpace`(A) returns the space generated by the rows of A, disregarding the domain of A. Therefore the row space may be of higher dimension than the image.

Example:

```
> RowSpace(A);
Vector space of degree 6, dimension 3 over GF(3^2)
Echelonized basis:
(  1   0   0   0 g^5 g^2)
(  0   1   0   0   0   0)
(  0   0   0   1   0   g)
```

The dimension of the row space indicates that the rank of a is 3, but this may be verified using the function `Rank`.

```
> Rank(A);
3
```

It is possible to find the echelon form of a using `EchelonForm`(A). This function returns two values: the echelon form E and a matrix B (a product of elementary matrices) such that $BA = E$.

```
> E,B:=EchelonForm(A);
> E,B;
[  1   0   0   0 g^5 g^2]
[  0   1   0   0   0   0]
[  0   0   0   1   0   g]
[  0   0   0   0   0   0]

[  0   0   2 g^6]
[  1   0   0   0]
[  0   0   0 g^7]
[  0   1   0   0]
> B*A eq E;
true
```

Also available are `Kernel`(A) (or `NullSpace`(A)) and `Cokernel`(A). The kernel is the subspace of the domain which is mapped to zero, and the cokernel is the subspace of the codomain which is not in the image.

Example:

```
> Kernel(A);
Vector space of degree 4, dimension 0 over GF(3^2)
> Cokernel(A);
Full Vector space of degree 1 over GF(3^2)
Mapping from: ModTupFld: W to Full Vector space of degree 1 over GF(3^2)
> Dimension(Kernel(A))+Dimension(Image(A)) eq Dimension(Domain(A));
true
> Dimension(Cokernel(A))+Dimension(Image(A)) eq Dimension(Codomain(A));
true
```

10.4 Row and Column Operations

The three elementary row and column operations on matrices are available as procedures in MAGMA . The following table shows them.

MAGMA	Meaning
`AddRow(~a, u, i, j)`	Adds u times row i to row j in matrix a
`AddColumn(~a, u, i, j)`	Adds u times column i to row j in matrix a
`MultiplyRow(~a, u, i)`	Multiplies row i of matrix a by u
`MultiplyColumn(~a, u, i)`	Multiplies column i of matrix a by u
`SwapRows(~a, i, j)`	Interchanges rows i and j of matrix a
`SwapColumns(~a, i, j)`	Interchanges columns i and j of matrix a

Each procedure destructively modifies the given matrix, so a $\sim$ must precede its identifier name in the procedure call.

Example:
In the following example, a copy $A1$ of A is made, and then g times row 1 of $A1$ is added to row 4, and columns 2 and 5 are swapped.

```
> A1:=A;
> AddRow(~A1,g,1,4);
> A1;
[  0   1   0   0   0   0]
[  0   0   0   0   0   0]
[  2   0   0 g^7   g g^5]
[  0   g   0   g   0 g^2]
> SwapColumns(~A1,2,5);
> A1;
[  0   0   0   0   1   0]
[  0   0   0   0   0   0]
[  2   g   0 g^7   0 g^5]
[  0   0   0   g   g g^2]
```

10.5 Simultaneous Systems of Linear Equations

MAGMA provides an easy way of solving a simultaneous system of linear equations that is expressed as a matrix equation $VX = W$, where X is a matrix, W is a vector with the same number of entries as X has columns, and V is an unknown vector with the same number of entries as X has rows. The function `Solution`(X, W) returns a particular solution v, and also returns the kernel K of X, regarding X as a homomorphism from the vector space of V to the vector space of W. The kernel is the space of vectors which when multiplied by X equal the zero vector, so the general solution to $VX = W$ is $V = v + k$ where k is any vector in K.

Example:
Suppose the user wants to solve the system

$$\begin{aligned} a + 2b + 2c - 3d &= 5 \\ 2a - b + c + 2d &= 1 \\ 3a + b + 3c - d &= 6 \end{aligned}$$

which is equivalent, once the vectors are written as row vectors, to

$$\begin{pmatrix} a & b & c & d \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 2 & -1 & 1 \\ 2 & 1 & 3 \\ -3 & 2 & -1 \end{pmatrix} = \begin{pmatrix} 5 & 1 & 6 \end{pmatrix}$$

In MAGMA this system may be solved as follows.

```
> Q := RationalField();
> V4 := VectorSpace(Q, 4);
> V3 := VectorSpace(Q, 3);
> H43 := Hom(V4, V3);
> X := H43![1, 2, 3, 2, -1, 1, 2, 1, 3, -3, 2, -1];
> X;
[ 1  2  3]
[ 2 -1  1]
[ 2  1  3]
[-3  2 -1]
> W := V3![5, 1, 6];
> v, K := Solution(X, W);
> v;
(   0    0 13/7 -3/7)
> K;
Vector space of degree 4, dimension 2 over Rational Field
Echelonized basis:
(   1    0 -8/7 -3/7)
(   0    1 -1/7  4/7)
```

The identifier v now contains a particular solution to the equation, and K contains the kernel of X.

```
> v*X eq W and Kernel(X) eq K;
   true
```

Therefore the general solution is

$$V = \left(0, 0, \frac{13}{7}, -\frac{3}{7}\right) + t_1 \left(1, 0, -\frac{8}{7}, -\frac{3}{7}\right) + t_2 \left(0, 1, -\frac{1}{7}, \frac{4}{7}\right)$$

If the user is not sure whether there is a solution to the equation $VX = W$, then `IsConsistent`(X, W) should be used instead. It returns **true** if a solution exists, and if so, then it also returns a particular solution and the kernel, just as `Solution` does.

Example:

```
> IsConsistent(X, V3![5, 1, 7]);
false
> IsConsistent(X, W);
true (   0    0 13/7 -3/7)
```

10.6 Changing the Basis

10.6.1 Defining the Chosen Basis

The function `VectorSpaceWithBasis`(B), where B is a sequence of linearly independent vectors, allows the user to define a chosen basis B for a vector space. (An alternative version of this function takes as its argument not a sequence but a matrix whose rows are the basis vectors.)

Example:
The following lines show how to build a vector space Sw of degree 3 over the rationals which has the basis $[[1, 0, 1], [2, 1, 1], [1, 1, 1]]$, as compared to the same vector space S with the standard basis.

```
  > Basis(S);
[
    (1 0 0),
    (0 1 0),
    (0 0 1)
]
> Sw:=VectorSpaceWithBasis([S | [1,0,1], [2,1,1], [1,1,1]]);
> Basis(Sw);
[
    (1 0 1),
    (2 1 1),
    (1 1 1)
]
```

The vector spaces S and Sw are considered by Magma to be the same in most respects. Note especially that when a vector is being created it makes no difference whether the sequence of its entries is coerced into S or Sw.

Example:

```
> S eq Sw;
> S![2, 5, 72] eq Sw![2, 5, 72];
true
```

The only time that the basis of Sw is significant, apart from when using the functions that access the basis and generators, is for the function `Coordinates`(V, v). Given a vector v in a vector space V, it returns the coordinates of v relative to the current basis of V.

Example:

```
> v := Sw![-2, -1, 2];
> v;
(-2 -1  2)
> Coordinates(Sw, v);
[ 3, -4, 3 ]
```

Observe that $v = 3(1,0,1) - 4(2,1,1) + 3(1,1,1)$.

To simulate a vector space whose entries are radically connected with a non-standard basis, the user should create a matrix which maps the vectors of the standard basis to those of the required basis.

Example:

```
> M := EndomorphismAlgebra(S);
> M;
Full Matrix Algebra of degree 3 over Rational Field
> m := M![1,0,1, 2,1,1, 1,1,1];
> mInv := m^-1;
> mInv;
[ 0  1 -1]
[-1  0  1]
[ 1 -1  1]
```

where the rows of m are the non-standard basis vectors. Now, given a vector, it may be multiplied on the right by `mInv` to make it into as a vector expressed relative to the other basis.

```
> S![-2, -1, 2]*mInv;
( 3 -4  3)
```

The output is a vector of the simulated vector space.

10.6.2 Constructing a Basis Gradually

MAGMA provides some functions to help the user construct linearly independent vectors. The simplest of these is `IsIndependent`(S), which returns **true** if the elements of the set or sequence S are linearly independent.

Example:

```
> vecseq := [V5 | [4/3, 8, -2, 0, 1/2], [5, 7, 10, -2, 0]];
> IsIndependent(vecseq);
true
```

The function `ExtendBasis`(Q, V) constructs a basis for a vector space V such that the linearly independent sequence of vectors Q is the first part of the resulting basis sequence.

Example:

```
> ExtendBasis(vecseq, V5);
[
    (4/3   8  -2   0 1/2),
    ( 5  7 10 -2  0),
    (0 0 1 0 0),
    (0 0 0 1 0),
    (0 0 0 0 1)
]
```

The space V need not be a full vector space. Another form of this function is `ExtendBasis`(U, V), which takes a subspace U of V as its first argument. The basis sequence that it constructs begins with the basis of U.

Part III

Coding Theory

Chapter 11

Error-Correcting Codes

The three main families of error-correcting codes supported in MAGMA are:

- Linear codes over finite fields;
- General (non-linear) codes over finite fields;
- Linear codes over finite euclidean rings, such as $\mathbb{Z}/n\mathbb{Z}$.

All of these are block codes, that is, codes in which each codeword has the same length. In this book, emphasis is given to linear codes over finite fields, so the reader should consult the *Handbook* for an account of non-linear codes and codes over rings.

The design of the MAGMA machinery for codes stresses the algebraic basis of coding theory. The facilities may be broadly grouped as follows: constructions; elementary operations; weight distribution properties; isomorphism/automorphism questions (monomial action and permutation action); and decoding algorithms.

Let R be a finite ring, and let R^n be the free module of rank n over R. The **Hamming distance** $d(u, v)$ between two vectors u and v of R^n is defined as the number of coordinate positions in which they differ. An (n, M, d) code C over R is a set of M vectors from R^n such that the distance between any two distinct members of C is at least d. An $[n, k, d]$ linear code C over R is a k-dimensional subspace of R^n such that the distance between any two distinct members of C is at least d. In this chapter, R will be a finite field $K = GF(q)$, unless stated otherwise.

11.1 Defining a Linear Code

In MAGMA , linear codes belong to the category `Code`. They are regarded as subspaces of K^n, with additional operations.

11.1.1 Defining a Code from a Vector Space

A linear code may be created by defining the appropriate subspace V of K^n and applying the transfer function `LinearCode`(V).

Example:
Consider the length-6 code C over $K = GF(3)$ generated by the vectors $(2, 0, 1, 2, 0, 1)$,

$(1,1,1,2,1,2)$, and $(1,2,0,2,1,0)$. It may be created in MAGMA as follows.

```
> K := GF(3);
> vsp := sub< VectorSpace(K, 6) |
> [2,0,1,2,0,1], [1,1,1,2,1,2], [1,2,0,2,1,0] >;
> vsp;
Vector space of degree 6, dimension 3 over GF(3)
Generators:
(2 0 1 2 0 1)
(1 1 1 2 1 2)
(1 2 0 2 1 0)
Echelonized basis:
(1 0 2 0 2 0)
(0 1 2 0 0 1)
(0 0 0 1 1 2)
>
> C := LinearCode(vsp);
> C;
[6, 3, 3] Linear Code over GF(3)
Generator matrix:
[1 0 2 0 2 0]
[0 1 2 0 0 1]
[0 0 0 1 1 2]
```

Notice from the output that the generator matrix of C is the (echelonized) basis matrix of the vector space `vsp`. C is a $[6,3,3]$ linear code, meaning that the number of components in each codeword is 6, the dimension of the vector space corresponding to the code is 3, and the minimum weight of the codewords is 3.

In general, a linear code over a finite field may be described as an $[n,k,d]$ code, where n is the block length, k is the dimension, and d is the minimum weight.

It is possible to circumvent the direct construction of the vector space by means of the `LinearCode` constructor:

`LinearCode<` K, n | generator specification `>`

It constructs the linear code generated by the vectors in K^n specified on the right side of the constructor; the generators may be given in any of the forms suitable for a `sub` constructor as applied to a vector space.

Example:
This example shows how to create the code C using this constructor.

```
> C2 := LinearCode< K, 6 |
> [2,0,1,2,0,1], [1,1,1,2,1,2], [1,2,0,2,1,0] >;
> C eq C2;
true
```

11.1.2 Defining a Code from a Generator Matrix

Another way to create a linear code in MAGMA is to specify its generator matrix. If A is a rectangular matrix over a finite field K, then `LinearCode`(A) returns the linear code generated by the rows of A (echelonized if necessary).

Example:
The code C described above could also be defined by creating a matrix whose rows are the generator vectors, and applying `LinearCode` to this matrix.

```
> M := KMatrixSpace(K, 3, 6);
> M;
Full KMatrixSpace of 3 by 6 matrices over GF(3)
> genmat := M ! [2,0,1,2,0,1, 1,1,1,2,1,2, 1,2,0,2,1,0];
> C3 := LinearCode(genmat);
> C eq C3;
true
```

11.2 Calculations with Codewords and Vectors

Because of the close relationship between linear codes and vector spaces, codewords in MAGMA are implemented as ordinary vectors. Given a code C of length n over K, a codeword of C may be created by coercing a length-n sequence (of integers or elements of K) into C. However, the parent of the result will not be C but K^n. *The parent magma is not the linear code, but the corresponding generic vector space*.

Example:
The codeword $(2, 0, 1, 1, 2, 2)$ of C may be assigned to c as follows.

```
> c := C ! [2, 0, 1, 1, 2, 2];
> c;
(2 0 1 1 2 2)
> Category(c);
ModTupFldElt
> Parent(c);
Full Vector space of degree 6 over GF(3)
```

An individual component of a codeword may be altered using index notation on the left side of an assignment statement; this operation imitates the effect of sending the codeword along a noisy communications channel.

Example:
The fourth component of the codeword c in C can be changed to 2 as follows.

```
> c[4] := 2;
> c;
(2 0 1 2 2 2)
```

Although linear codes do not act as parent magmas, membership and subset testing is still provided, by means of the operators `in` and `subset`.

Example:

```
> c in C;
false
```

Coercion of a vector (or a sequence representing a vector) into a code is only allowed if the vector is a member of the code. If the vector is not a codeword, an error message will be given.

Example:

```
> C!c;

>> C!c;
    ^
Runtime error in '!': Result is not in the given structure
```

The following table lists the functions on vectors that are designed for use in the context of linear codes. See also Section 10.3, for the general-purpose operations on vectors.

The function `Syndrome`(x, C) provides a way to calculate the syndrome of a vector x relative to a linear code C without explicitly having to calculate the parity check matrix of C first.

Example:
The fact that the syndrome of c relative to C is non-zero is another demonstration that the new version of c is not in the code C.

```
 > Syndrome(c, C);
(2 2 2)
```

The function `Support`(v) returns the set of coordinate positions for which a vector v has non-zero components. It is useful for such tasks as building designs from codes.

Example:
The support of c shows that only its second entry is zero.

```
> Support(c);
{ 1, 3, 4, 5, 6 }
```

11.3 General Facts about a Linear Code

The following table lists functions giving general information about a code

MAGMA	Meaning
`Alphabet(C)`, `Field(C)`	Underlying field K for code C
`Length(C)`	Block length n of $[n, k, d]$ linear code C
`Dimension(C)`	Dimension k of C
`MinimumWeight(C)`, `MinimumDistance(C)`	Minimum weight d of words in C
`StandardForm(C)`	Returns (i) a code S equivalent to C but in standard form, (i.e., formed from C with the columns permuted so that the information columns are columns $1, \ldots, k$) (ii) isomorphism $f : C \to S$. There are many such codes, but the same one is returned each time.
`GeneratorMatrix(C)`	Generator matrix for C
`ParityCheckMatrix(C)`	Parity check matrix for C
`InformationSpace(C)`	k-dimensional vector space of information vectors for C
`InformationSet(C)`	Length-k sequence giving the numbers of the information columns of C (i.e., k linearly independent columns of the generator matrix G, such that G is the identity matrix when restricted to these columns)
`AllInformationSets(C)`	All possible information sets of C, as a sorted sequence whose terms are length-k sequences of indices of linearly independent columns
`SyndromeSpace(C)`	$(n-k)$-dimensional vector space of syndrome vectors for C
`VectorSpace(C)`	Vector subspace whose basis corresponds to the rows of the generator matrix for C
`AmbientSpace(C)`	Generic vector space K^n
`Generic(C)`	Generic $[n, n, 1]$ code containing C
`Dual(C)`	Dual code of C

The following table lists tests for properties of the code.

MAGMA	Meaning
`IsCyclic(C)`	**true** if C is cyclic
`IsMDS(C)`	**true** if C is maximum-distance separable (i.e., is an $[n, k, n-k+1]$ code)
`IsSelfOrthogonal(C)`, `IsSelfDual(C)`	**true** if C is self-orthogonal
`IsWeaklySelfOrthogonal(C)`, `IsWeaklySelfDual(C)`	**true** if C is weakly self-orthogonal

In these tables, it should be understood that C is an $[n, k, d]$ linear code over the finite field K. Most of these functions are fairly trivial, with the exception of `MinimumWeight`(C); the advanced facilities for weight enumeration are discussed later in the chapter. There are some additional functions available for cyclic codes, listed later.

Example:
Suppose that the binary $[4, 2, 2]$ code generated by $(1, 0, 1, 0)$ and $(0, 1, 1, 1)$ is created in MAGMA with the following assignment statement.

```
> lc := LinearCode< GF(2), 4 | [1,0,1,0], [0,1,1,1] >;
```

While this one-statement code definition is attractively compact, it has the minor disadvantage that the algebraic structures auxiliary to `lc` are nameless. The functions listed in the table provide a means of accessing these structures.

Example:
If the user wishes to define the vector $v = (0, 0, 0, 1)$, which is not in `lc` but is in the vector space of which `lc` is a subspace, then the `AmbientSpace`(C) should be used to obtain this vector space.

```
> v := AmbientSpace(lc) ! [0, 0, 0, 1];
```

The access operations are also useful when writing a function whose argument is an arbitrary code. For instance, according to the sphere-packing bound, any q-ary $(n, M, 2t+1)$ code satisfies

$$M\left\{\binom{n}{0} + \binom{n}{1}(q-1) + \cdots + \binom{n}{t}(q-1)^t\right\} \leq q^n.$$

If equality is achieved then the code is called **perfect**.

The generators of a code C are stored in echelonized form, which may differ from the way in which they were given by the user. As usual, `C.i` returns the i-th generator, and `Generators`(C) returns the generators in a set. Another function, `GeneratorMatrix`(C), returns a matrix whose rows are the generators.

Example:

```
> g := GeneratorMatrix(lc);
> g;
[1 0 1 0]
[0 1 1 1]
> h := ParityCheckMatrix(lc);
> h;
[1 0 1 1]
[0 1 0 1]
> g * Transpose(h);
[0 0]
[0 0]
```

11.4 Families of Linear Codes

MAGMA has functions which return many of the well-known linear codes: Hamming, Reed-Muller, Golay, Reed-Solomon, cyclic, alternant, and generalized BCH. For all of these well-known codes, it is much more efficient to define them using their special functions rather than specifying their generators by hand.

The following table summarizes the functions which return some of the basic linear codes. In the table, q denotes the cardinality of the field K.

Example:
The binary Hamming code $\mathcal{H}_{2,3}$ is created by the assignment statement.

```
> h3:=HammingCode(GF(2),3);
> h3;
[7, 4, 3] "Hamming code (r = 3)" Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 1 0]
[0 1 0 0 0 1 1]
[0 0 1 0 1 1 1]
[0 0 0 1 1 0 1]
```

11.4.1 Cyclic Codes

The following table lists the functions for the creation of cyclic codes.

MAGMA	Meaning
`CyclicCode(u)`	$[n, k]$ cyclic code generated by the right cyclic shifts of vector $u \in K^n$
`CyclicCode(n, f)`	Length-n cyclic code generated by the right cyclic shifts of the vector corresponding to the polynomial $f \in K[x]$, where f has degree $n-k$ and divides $x^n - 1$
`CyclicCode(n, R, K)`	Length-n cyclic code over K generated by the least-degree polynomial having the elements of R as roots, where R is a set or sequence of primitive n-th roots of unity in an extension of K
`BCHCode(K, n, d, b)`	q-ary $[n, k \geq n - m(d-1), D \geq d]$ BCH code of designated distance d having generator polynomial $g(x) = \text{lcm}\{m_1(x), \ldots, m_{d-1}(x)\}$, where $m_i(x)$ is the minimum polynomial of α^{b+i-1}, for $i = 1, \ldots, d-1$; here b is a positive integer, α is a primitive n-th root of unity in the degree-m extension of K (i.e., $GF(q^m)$), and n must satisfy $\gcd(n, q) = 1$
`BCHCode(K, n, d)`	As above but with $b = 1$
`QRCode(K, n)`	q-ary length-n quadratic residue code with generator polynomial $g_0(x) = \prod(x - \alpha^r)$, where α is a primitive nth root of unity in an extension field L of K, and the product is taken over the quadratic residues in L; here n is an odd prime such that q is a quadratic residue modulo n

The special operations on cyclic codes are given in the table below; they may be applied to any code which is cyclic.

MAGMA	Meaning
`GeneratorPolynomial(C)`	A generator polynomial for cyclic code C
`CheckPolynomial(C)`	A check polynomial for cyclic code C
`Idempotent(C)`	Idempotent for cyclic code C

Example:
The length-7 binary cyclic code generated by the vector $(1, 1, 0, 1, 0, 0, 0)$ corresponding to

the polynomial $1 + x + x^3$ can be generated in any of the following ways.

```
> P<x> := PolynomialRing(GF(2));
> cyc := CyclicCode(7, 1+x+x^3);
> cyc;
[7, 4, 3] "Hamming code (r = 3)" Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 1 0]
[0 1 0 0 0 1 1]
[0 0 1 0 1 1 1]
[0 0 0 1 1 0 1]

> v := VectorSpace(GF(2), 7) ! [1,1,0,1,0,0,0];
> cyc2 := CyclicCode(v);
> cyc2 eq cyc;
true

> L<w> := GF(2, 3);
> cyc3 := CyclicCode(7, {w, w^2, w^4}, GF(2));
> cyc3 eq cyc;
true
```

Example:
The length-15 binary BCH code with $b = 1$ which is 2-error-correcting and hence has designated distance 5 is

```
> BCH15 := BCHCode(GF(2), 15, 5);
> BCH15;
[15, 7, 5] "BCH code (d = 5, b = 1)" Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 1 0 0 0 1 0 1 1]
[0 1 0 0 0 0 0 1 1 0 0 1 1 1 0]
[0 0 1 0 0 0 0 0 1 1 0 0 1 1 1]
[0 0 0 1 0 0 0 1 0 1 1 1 0 0 0]
[0 0 0 0 1 0 0 0 1 0 1 1 1 0 0]
[0 0 0 0 0 1 0 0 0 1 0 1 1 1 0]
[0 0 0 0 0 0 1 0 0 0 1 0 1 1 1]
```

11.5 Constructing Codes from Other Codes

There are many ways of constructing a new code by modifying the words of an existing code in some way. One of these is to build a subcode, with the usual sub constructor. In addition, MAGMA has functions corresponding to the constructions specific to coding theory. The following table summarizes these functions; in the table, C should be understood to be an $[n, k, d]$ linear code.

MAGMA	Meaning
`AugmentCode(C)`	Code consisting of the words of binary code C and the all-ones vector (if it is not already in C)
`ExtendCode(C)`	Code formed from binary code C by adding an overall parity check (i.e., adding a 0 at the end of every even-weight codeword, and a 1 at the end of every odd-weight codeword)
`ExpurgateCode(C)`	Code formed from binary code C by deleting all odd-weight codewords
`LengthenCode(C)`	Code formed from binary code C by augmenting then extending it
`DirectSum(C, D)`	Direct sum of codes C and D over same field, (i.e., all vectors $u\|v$, where $u \in C$, $v \in D$)
`PunctureCode(C, i)`	Given binary code C and $i \in \{1, \ldots, n\}$, forms code with i-th component of each codeword deleted
`PunctureCode(C, S)`	Given binary code C and set $S \subseteq \{1, \ldots, n\}$, forms code with codeword components indexed by S deleted
`ShortenCode(C, i)`	Given binary code C and $i \in \{1, \ldots, n\}$, forms code containing only those codewords with zero in i-th component and with this component deleted from them
`ShortenCode(C, S)`	Given binary code C and set $S \subseteq \{1, \ldots, n\}$, forms code containing only those codewords with zero in each component indexed by S and with these component deleted from them
`SubfieldSubcode(C, S)`	Given code C over K and subfield S of K, returns (i) code consisting of those words of C whose components are in S (ii) restriction map
`SubfieldSubcode(C)`	As above, where S is the prime subfield of K

For binary codes, one important characteristic of a codeword is its parity, that is, whether its weight is odd or even. Even-weight codewords are particularly important because any linear combination of such words is also even. Given a binary linear code C, the function `ExpurgateCode`(C) returns the code which contains only those codewords of C which are even.

Example:

```
> h4 := HammingCode(GF(2), 4);
> h4;
[15, 11, 3] "Hamming code (r = 4)" Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 1 1 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 1 1]
[0 0 0 1 0 0 0 0 0 0 0 1 1 0 1]
[0 0 0 0 1 0 0 0 0 0 0 1 0 1 0]
[0 0 0 0 0 1 0 0 0 0 0 0 1 0 1]
[0 0 0 0 0 0 1 0 0 0 0 1 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 1 1 1]
[0 0 0 0 0 0 0 0 1 0 0 1 1 1 1]
```

```
[0 0 0 0 0 0 0 0 0 1 0 1 0 1 1]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 1]
> h4exp:=ExpurgateCode(h4);
> h4exp;
[15, 10, 4] Cyclic Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 1 0 1 0 1]
[0 1 0 0 0 0 0 0 0 0 1 1 1 1 1]
[0 0 1 0 0 0 0 0 0 0 1 1 0 1 0]
[0 0 0 1 0 0 0 0 0 0 0 1 1 0 1]
[0 0 0 0 1 0 0 0 0 0 1 0 0 1 1]
[0 0 0 0 0 1 0 0 0 0 1 1 1 0 0]
[0 0 0 0 0 0 1 0 0 0 0 1 1 1 0]
[0 0 0 0 0 0 0 1 0 0 0 0 1 1 1]
[0 0 0 0 0 0 0 0 1 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 0 1 0 1 0 1 1]
```

Another way of constructing a code in which every word is even is to use all the words of the given code but to modify them so that they are even. The `ExtendCode`(C) function does this, by extending each word with a parity-check digit. If the minimum weight d is odd, the new code will be an $[n+1, k, d+1]$ code.

Example:

```
> h4ext := ExtendCode(h4);
> h4ext;
[16, 11, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1]
[0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1]
[0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0]
[0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1]
[0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
```

Note that the generator matrix for `h4ext` is the same as the one for `h4`, except that a column has been added on the right so as to give each generator codeword even parity.

If a binary linear code C does not contain the all-ones vector then the function `AugmentCode`(C) will return a larger linear code which contains C, by including the all-ones vector with the words of C.

Example:
Every binary Hamming code contains the all-ones vector of the relevant length already, so this function would merely return `h4` itself if applied to `h4`, but the dual of `h4` will give a more fruitful result.

```
> dualh4 := Dual(h4);
> dualh4;
[15, 4, 8] Cyclic Linear Code over GF(2)
Generator matrix:
[1 0 0 0 1 0 0 1 1 0 1 0 1 1 1]
[0 1 0 0 1 1 0 1 0 1 1 1 1 0 0]
[0 0 1 0 0 1 1 0 1 0 1 1 1 1 0]
[0 0 0 1 0 0 1 1 0 1 0 1 1 1 1]
> dualh4aug := AugmentCode(dualh4);
> dualh4aug;
[15, 5, 7] Cyclic Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 1 1 0 1 1 0 0 1 0]
[0 1 0 0 0 0 1 1 1 0 1 1 0 0 1]
[0 0 1 0 0 1 1 0 1 0 1 1 1 1 0]
[0 0 0 1 0 0 1 1 0 1 0 1 1 1 1]
[0 0 0 0 1 1 1 0 1 1 0 0 1 0 1]
> dualh4 subset dualh4aug;
true
```

The last line above confirms that `dualh4` is a subcode of `dualh4aug`.

The `LengthenCode`(C) function does two of the above operations in one step: it first augments C with the all-ones vector and then extends the new code with a parity-check.

Example:
This may be tested as follows.

```
>  LengthenCode(dualh4) eq ExtendCode(dualh4aug);
true
```

There are also some functions which construct new codes according to properties of individual components of codewords. `PunctureCode`(C, i) deletes the i-th component of each codeword of C to form a new code. The puncturing operation can also be performed on more than one coordinate, so there is an alternative form of the function, `PunctureCode`(C, S), where S is the set of component coordinates to be deleted.

The `ShortenCode`(C, i) function differs from `PunctureCode` in that only the codewords which have a zero in the i-th component are selected for the new code. Then these codewords have this component deleted. If a code is to be shortened in several components then the appropriate function is `ShortenCode`(C, S), where S is the set of components.

Example:
$\mathcal{H}_{11,2}$ may be shortened in the first and second positions:

```
h11;
[12, 10, 3] "Hamming code (r = 2)" Linear Code over GF(11)
Generator matrix:
[ 1  0  0  0  0  0  0  0  0  0  1 10]
[ 0  1  0  0  0  0  0  0  0  0  1  9]
[ 0  0  1  0  0  0  0  0  0  0  2  8]
[ 0  0  0  1  0  0  0  0  0  0  3  7]
[ 0  0  0  0  1  0  0  0  0  0  4  6]
[ 0  0  0  0  0  1  0  0  0  0  5  5]
[ 0  0  0  0  0  0  1  0  0  0  6  4]
[ 0  0  0  0  0  0  0  1  0  0  7  3]
[ 0  0  0  0  0  0  0  0  1  0  8  2]
[ 0  0  0  0  0  0  0  0  0  1  9  1]
ShortenCode(h11, {1, 2});
[10, 8, 3] Linear Code over GF(11)
Generator matrix:
[ 1  0  0  0  0  0  0  0  2  8]
[ 0  1  0  0  0  0  0  0  3  7]
[ 0  0  1  0  0  0  0  0  4  6]
[ 0  0  0  1  0  0  0  0  5  5]
[ 0  0  0  0  1  0  0  0  6  4]
[ 0  0  0  0  0  1  0  0  7  3]
[ 0  0  0  0  0  0  1  0  8  2]
[ 0  0  0  0  0  0  0  1  9  1]
```

11.6 The Weight Distribution

The most important invariant of a code C is its minimum distance. In a linear code, this corresponds to the minimum weight of the codewords of C, where the (Hamming) weight of a codeword is defined as the number of non-zero components. The following table lists some of MAGMA 's functions connected with minimum weight and weight distribution in a code.

MAGMA	Meaning
`CoveringRadius(C)`	Covering radius of C
`Diameter(C)`	Largest weight of words of C
`MinimumWeight(C)`, `MinimumDistance(C)`	Minimum weight d of words in C
`Weight(u)`	Weight of vector u
`MinimumWords(C)`	Set of words of C having minimum weight
`Words(C, t)`	Set of words of C with weight t
`NumberOfWords(C, t)`	Number of words of C with weight t
`WeightDistribution(C)`	Sequence of tuples $\langle w_i, n_i \rangle$ where n_i is the number of codewords of C with weight w_i ($n_i > 0$)
`WeightDistribution(C, u)`	Weight distribution of coset $C + u$, giving weight-counts of coset vectors
`DualWeightDistribution(C)`	Weight distribution of dual of C
`MacWilliamsTransform(n,k,q,W)`	Supposing a hypothetical $[n, k]$ code C over $\mathbb{F}_q$, with weight distribution W, return weight distribution of dual of C

It should be noted that the determination of the minimum weight is, in general, a very expensive computation, although it can be performed without enumerating the words of the code. However, a more efficient algorithm exists for cyclic codes, so in that case significantly larger codes can be handled. For the determination of the complete weight distribution, it is necessary to enumerate all the codewords up to scalar multiples. Hence this calculation is much more expensive than that of finding the minimum weight.

The function `WeightDistribution`(C) may be used to determine the number of codewords of C of each weight. It returns a sequence of tuples, each of which is an integer pair such that the first number is a i-th weight and the second number is the number of codewords with that weight.

Example:

```
> WeightDistribution(h3);
[ <0, 1>, <3, 7>, <4, 7>, <7, 1> ]
> WeightDistribution(h4);
[ <0, 1>, <3, 35>, <4, 105>, <5, 168>, <6, 280>, <7, 435>, <8, 435>, <9, 280>,
<10, 168>, <11, 105>, <12, 35>, <15, 1> ]
```

The other MAGMA functions associated with weight distribution, including `MacWilliamsTransform`, express the distribution in the same form.

Information about weight distribution may be obtained in a different form from the function `WeightEnumerator`(C). It returns an integer polynomial in two indeterminates, x and y say, such that the coefficient of $x^i y^{n-i}$ is the number of codewords with weight i. There are two methods for naming the indeterminates (i.e., as identifiers and as printnames): either by creating the parent ring over the integers with generator assignment or by using generator assignment while assigning the polynomial. Both techniques are demonstrated below, so that the user may choose whichever is preferred.

Example:

```
> WeightEnumerator(h3);
$.1^7 + 7*$.1^4*$.2^3 + 7*$.1^3*$.2^4 + $.2^7
```

1st method:

```
> P<x,y> := PolynomialRing(IntegerRing(), 2);
> P!WeightEnumerator(h3);
x^7 + 7*x^4*y^3 + 7*x^3*y^4 + y^7
```

2nd method:

```
> wtenum<x,y> := WeightEnumerator(h3);
> wtenum;
x^7 + 7*x^4*y^3 + 7*x^3*y^4 + y^7
```

The weight enumerator functions are listed in the table below.

MAGMA	Meaning
`WeightEnumerator(C)`	Polynomial in two indeterminates in which coefficient of $x^i y^{n-i}$ is number of codewords of C with weight i
`WeightEnumerator(C, u)`	Weight enumerator of coset $C + u$, giving weight-counts of coset vectors as a polynomial (of form described above)
`CompleteWeightEnumerator(C)`	Complete weight enumerator of C over $\mathbb{F}_q$, as a polynomial in q indeterminates
`CompleteWeightEnumerator(C, u)`	Complete weight enumerator of coset $C + u$
`MacWilliamsTransform(n,k,K,W)`	Supposing a hypothetical $[n, k]$ code C over K, with complete weight enumerator W, return complete weight enumerator of dual of C

11.7 Equivalent and Isomorphic Codes

Two linear codes in K^n are equivalent if one may be obtained from the other by a monomial action. The function `IsEquivalent`(C, D) tests whether the codes C and D are equivalent. If they are equivalent, it returns the corresponding mapping from C to D as a second value.

Example:

```
> C1 := LinearCode<GF(3), 3 | [1,0,1]>;
> C2 := LinearCode<GF(3), 3 | [1,2,0]>;
> i, f := IsEquivalent(C1, C2);
> i;
true
```

```
> f(C1) eq C2;
true
```

Two linear codes in K^n are isomorphic if one may be obtained from the other by a permutation action, that is, by permuting the columns (coordinate positions). For binary codes, this is the same as equivalence, but for other codes, isomorphism is a stronger relationship than equivalence. Currently, isomorphism testing is only possible in MAGMA for binary codes. The function `IsIsomorphic`(C, D) tests whether the binary codes C and D are isomorphic; if the principal return value is true, it returns as a second value a mapping which is an isomorphism from C to D.

Example:
Consider the first-order binary Reed-Muller code R with $m = 4$, and the code T formed by the action of $p = (4, 13)(5, 12)(6, 11)$ under R. The following lines demonstrate that R and T are isomorphic, as expected.

```
> R := ReedMullerCode(1, 4);
> R;
[16, 5, 8] "Reed-Muller Code (r = 1, m = 4)" Linear Code over GF(2)
Generator matrix:
[1 0 0 1 0 1 1 0 0 1 1 0 1 0 0 1]
[0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1]
[0 0 1 1 0 0 1 1 0 0 1 1 0 0 1 1]
[0 0 0 0 1 1 1 1 0 0 0 0 1 1 1 1]
[0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1]
> p := Sym(16) ! (4, 13)(5, 12)(6, 11);
>  T := R ^ p;
> T;
[16, 5, 8] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 1 0 1 0 1 0 1 1 1 1 0]
[0 1 0 0 0 1 1 0 1 0 0 1 1 1 0 1]
[0 0 1 0 0 0 0 0 1 1 1 1 1 0 1 1]
[0 0 0 1 0 0 1 1 0 0 1 1 0 1 1 1]
[0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0]
> R eq T;
false
> iso, m := IsIsomorphic(R, T);
> iso;
true
> m;
Mapping from: Full Vector space of degree 16 over GF(2) to Full Vector space
of degree 16 over GF(2)
```

Note that isomorphic codes need not have the same standard form. The function `StandardForm`(C) returns a code in standard form that is isomorphic to C. It returns the same code every time, but there are actually many codes satisfying the description.

Example:

```
> StandardForm(R) eq StandardForm(T);
false
```

11.8 Encoding and Decoding: An example

The original motivation for coding theory was the encoding and decoding of information. This section will demonstrate the concepts involved, and their associated functions.

11.8.1 Encoding

Consider the extended binary Hamming code $C = \widehat{\mathcal{H}_{2,4}}$. This code may be created in MAGMA by extending $\mathcal{H}_{2,4}$ and putting the result in standard form.

```
> h4 := HammingCode(GF(2), 4);
> C := StandardForm(ExtendCode(h4));
> C;
[16, 11, 4] Linear Code over GF(2)
Generator matrix:
[1 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1]
[0 1 0 0 0 0 0 0 0 0 0 0 1 1 0 1]
[0 0 1 0 0 0 0 0 0 0 0 0 0 1 1 1]
[0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 0]
[0 0 0 0 1 0 0 0 0 0 0 1 0 1 0 1]
[0 0 0 0 0 1 0 0 0 0 0 0 1 0 1 1]
[0 0 0 0 0 0 1 0 0 0 0 1 1 1 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 1 1 1 0]
[0 0 0 0 0 0 0 0 1 0 0 1 1 1 1 1]
[0 0 0 0 0 0 0 0 0 1 0 1 0 1 1 0]
[0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 1]
```

Since the code C has length 16 and dimension 11, it encodes messages of 11 binary digits into codewords of 16 binary digits. The minimum weight of C is 4, so it can correct any single error, using nearest-neighbour decoding, and detect any double error.
Suppose now that the message to be encoded is $(0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1)$. It may be defined in MAGMA as a vector in the information space of C.

```
> infospace := InformationSpace(C);
> infospace;
Full Vector space of degree 11 over GF(2)
> message := infospace![0,1,1,0,1,0,0,1,1,0,1];
```

To encode this vector, it must be multiplied by the generator matrix of C.

```
> encoded := message * GeneratorMatrix(C);
> encoded;
(0 1 1 0 1 0 0 1 1 0 1 1 1 1 0 1)
```

Observe that `encoded` is the message vector followed by five check digits. This is because the code was created in standard form, in which the information set corresponds to the first k components of the $[n, k]$ code. This may be verified by typing

```
> InformationSet(C);
[ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 ]
```

to obtain the sequence from 1 to 11 inclusive. (The function returns a sequence rather than a set because it is often useful for the information set to be ordered.)

11.8.2 Error in Transmission

Now imagine that encoded is sent along a noisy communications channel, and that consequently an error occurs in the eighth place of the codeword. The following assignment statements will simulate this:

```
> received := encoded;
> received[8];
1
> received[8]:=0;
```

The task of decoding involves the detection and correction of the error in `received`.

11.8.3 Elementary Syndrome-Decoding Techniques

The syndrome of a vector relative to a code is zero if and only if that vector is a word of the code. A call to the function `Syndrome` will show that the syndrome of `received` is non-zero.

```
> syn := Syndrome(received, C);
> syn;
(1 1 0 1 0)
```

Therefore `received` is not a codeword. That is, the presence of error(s) in transmission has been detected.
Syndrome decoding rests on the theorem that two vectors have the same syndrome if and only if they are in the same coset of the code. The function `CosetLeaders`(C) returns a sequence containing all the coset leaders of a code C, and also returns a map from the syndrome space to the ambient space of the code. It is the map that is needed in this circumstance, because in practice it returns the coset leader with the given syndrome. The map is applied to the syndrome in the lines below:

```
> cl, syntoleader := CosetLeaders(C);
> errorvec := syn @ syntoleader;
> errorvec;
(0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0)
```

The error vector is all zero except for the eighth component, as expected. The received vector must be corrected by subtracting the error vector:

```
> corrected := received - errorvec;
```

Now the decoding process is almost complete, since the decoded message can be extracted as the information part of corrected, using the function `Coordinates`(C, u).

```
> decoded := infospace ! Coordinates(C, corrected);
> decoded eq message;
true
```

The message has been decoded successfully.

The limitations of a code can also be explored in this way. Suppose the received vector has an error in the sixth place as well. In this case, when the error vector is calculated using the same method as above, the answer is wrong; it should be $(0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0)$.

```
> rec2 := received;
> rec2[6] := 1;
> err2 := Syndrome(rec2, C) @ syntoleader;
> err2;
(1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0)
```

If the decoding process is continued with this incorrect error vector, then the decoded vector `dec2` will not equal the original message.

```
> corr2 := rec2 - err2;
> dec2 := infospace ! Coordinates(C, corr2);
> dec2;
(1 1 1 0 1 1 1 0 1 0 1)
```

The problem in this second example occurred because `rec2` was equally spaced (with a distance of two) between two codewords, `corrected` and `corr2`.

```
> Distance(rec2, corrected);
2
> Distance(rec2, corr2);
2
```

The decoding procedure regrettably chose the wrong codeword. Here is an example of a double error which C can detect but not correct.