A Brief Magma Tutorial

David R. Kohel

School of Mathematics and Statistics
The University of Sydney

Mathematical Sciences Research Institute
30 July 2006

What is Magma*?*

Magma is both a computer algebra system and a programming language.

• **Magma** commands are interpretted rather than compiled for dynamic interaction in a shell (analogous to perl or python).

• **Magma** makes available a huge library of mathematical datastructures together with high performance algorithms for their manipulation.

• **Magma** code, can be written in the **Magma** language as **packages** can be attached by users at startup time to expand on the functionality.

Features include algorithms for group theory, noncommutative algebra, commutative algebra, number theory, and algebraic geometry, etc.

Magma and Applications

1. *The Magma shell.*
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§1 The Magma shell

The most typical way to run `Magma` is interactively via the `Magma` shell. Every statement ends in a semicolon. Output not assigned to a variable, using `:=`, is printed to the standard output. `$1`, `$2`, and `$3` refer three previous objects sent to standard output.

```
chipotle:~> magma
Magma V2.11-12    Sun Jan 30 2005 18:46:41 on chipotle
Type ? for help.  Type <Ctrl>-D to quit.

Loading startup file "/home/kohel/.magma"

> 1;
1
> 2;
2
> $1; $2;
2 1
```

§1 The Magma shell [cont]

Notice that the `Magma` language can be expanded by users by automatically loading additional code (or default preferences) at startup. A startup file can be specified with the `MAGMA_STARTUP_FILE` environment variable.

E.g. in `csh` or `tcsh`:

```
setenv MAGMA_STARTUP_FILE /home/kohel/.magma
```

or in `bash`:

```
export MAGMA_STARTUP_FILE='/home/kohel/.magma'
```

§1 The Magma shell [cont]

Syntax. The assignment operator `:=` is used to assign the value on the right to the variable name on the left:

```
> x := 2;
> y := 3/4;
```

Every statement in `Magma` must end with a semicolon "`;`". A `Magma` statement may extend over several lines:

```
> x := 2 *
> 3 * 5 * 7
> ;
> x;
210
```

Note that `x;` and `print x;` give the same result.

Magma and Applications

1. The Magma shell.
2. *Parents and categories.*
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§2 **Parents and categories**

Every object in `Magma` has a `Parent` structure to which it belongs. Generally it is necessary to define the parent structure before initializing an element.

```
> QQ := RationalField();
> x := 2*One(QQ);
> x;
2
> Parent(x);
Rational Field
> IsUnit(x);
true
```

§2 **Parents and categories [cont]**

Note that the same construction with the integer ring produces a different element whose membership in $\mathbb{Z}$ rather than $\mathbb{Q}$ necessarily gives it different properties.

```
> ZZ := IntegerRing();
> y := 2*One(ZZ);
> y;
2
> Parent(y);
Integer Ring
> IsUnit(y);
false
```

The boolean function `IsUnit` must address 2 as an element of $\mathbb{Z}$, and since there is no element $1/2 \in \mathbb{Z}$, returns `false`.

§2 **Parents and categories [cont]**

Every object in `Magma` has an associated `Category` or `Type`. This is distinct from the concept of `Parent`, and analogous to the concept of a mathematical category (e.g. of rings, groups, or sets):

```
> Parent(x);
Rational Field
> Parent(x) eq QQ;
true
> Type(x);
FldRatElt
> Type(QQ);
FldRat
```

• The category handle can be used for comparisons (with `eq`) of possibly imcompatible objects, and for type checking, permitting function overloading.
• The formalism of the parent–element relationship facilitates the creation of maps between parents, which can be applied to elements.

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. *Primitive structures.*
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§3 **Primitive structures**

Certain categories, such as the `Integers()` and the `RationalField()` (with the operator `/` as an element constructor), are predefined as system-wide global structures, and do not have to be constructed in which to create elements.

```
> n := 2^127-1;
> n;
170141183460469231731687303715884105727
> r := 2/31;
> r;
2/31
> Type(r);
FldRatElt
> Parent(r);
Rational Field
```

Note that we haven't formally created any parent structure in order to create these elements. The parent object is global and a pointer to it automatically set up.

§3 **Primitive structures [cont]**

Other examples are the monoid of `Strings()`

```
> s := "Integer Ring";
> s;
Integer Ring
> Type(s);
MonStgElt
```

and the algebra of `Booleans` ({`true`, `false`}):

```
> true;
true
> true xor false;
true
> true and false;
false
```

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. *Aggregate structures.*
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§4 **Aggregate structures**

A. Sequences. A sequence is an indexed list of elements all of which have the same parent, called the `Universe` of the sequence. A common pitfall is to construct empty sequences without defining the universe.

```
> [];
[]
> Universe($1);

>> Universe($1);
           ^
Runtime error in 'Universe': Illegal null sequence
> [ ZZ | ];
[]
> Universe($1);
Integer Ring
```

§4 **Aggregate structures [sequences]**

If the universe of a sequence is not explicitly defined, then objects will be `coerced` into a common structure, if possible.

```
> S := [ 1, 2/31, 17 ];
> S;
[ 1, 2/31, 17 ]
> Universe(S);
Rational Field
> S[3];
17
> Parent($1);
Rational Field
```

§4 Aggregate structures [sequences]

The full syntax for sequence construction is:

```
[ Universe | Element : Loop | Predicate ]
```

As an example, we have the following sequence:

```
> FF<w> := FiniteField(3^6);
> [ FF | x : x in FiniteField(3^2) | Norm(x) eq 1 ];
[ 1, w^182, 2, w^546 ]
```

N.B. The finite fields $\mathbb{F}_3$, $\mathbb{F}_3^2$, and $\mathbb{F}_{3^6}$ are the unique finite fields of size 3, 3^2, and 3^6 (up to isomorphism). In one line, we have enumerated the four elements of the kernel of the norm map $\mathbb{F}_{3^2}^* \to \mathbb{F}_3^*$, and coerced these elements into the larger field $\mathbb{F}_{3^6}$. `Magma` has a sophisticated system for choosing compatible towers of embeddings of finite fields $\mathbb{F}_{p^n} \to \mathbb{F}_{p^{nm}}$.

§4 **Aggregate structures [sets]**

B. Sets. A set is an unordered collection of objects having the same parent, again, defined to be its `Universe`.

```
> { FiniteField(2^8) | 1, 2, 3, 4 };
{ 1, 0 }
> Random($1);
0
```

The syntax for set construction is analogous to that for sequences:

```
{ Universe | Element : Loop | Predicate }
```

The enumeration operator `#` applies to both `sequences` and `sets`.

```
> #[ x^2 : x in FiniteField(3^3) | x ne 0 ];
26
> #{ x^2 : x in FiniteField(3^3) | x ne 0 };
13
```

§4 **Aggregate structures [indexed sets]**

C. Indexed sets. An indexed set is a collection of objects indexed by the positive integers. An element is assigned the next available index at its first occurence.

```
> S := {@ 4, 3, 7 @};
> S;
{@ 4, 3, 7 @}
> T := {@ 1, 1, 11 @};
> S join T; /* Union operator. */
{@ 4, 3, 7, 1, 11 @}
> $1[4];
1
> #$2;
5
```

Indexed sets have advantages of fast hashed lookup (with the operator `in` or the function `Index`) on top of the indexing.

§4 **Aggregate structures [tuples]**

D. Tuples. A `tuple` is analogous to a `sequence`, but unlike sets and sequences, the parent structure – the set-theoretic product of the parents of the entries – stores the parent of each component.

```
> <>;
<>
> Parent($1);
Cartesian Product<>
> <1,2/1>;
<1, 2>
> Parent($1);
Cartesian Product<Integer Ring, Rational Field>
```

The parent structure of a `tuple` is more important than in the case of sequences or sets.

```
> C := CartesianProduct(Integers(),RationalField());
> t := C!<1,1>;
> Parent(t[2]);
Rational Field
```

§4 **Aggregate structures [vectors]**

E. Vectors. Since there is a unique global free module R^n of rank n over any ring R, with endomorphism algebra $M_n(R)$, the following shorthand constructor for vectors is provided.

```
> Vector([2,11,7]);
( 2 11 7)
```

Note that in contrast to tuples, which are the set-theoretic product R^n, as a module, elements of R^n support scalar multiplication by elements of R and addition.

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. *Element creation and transmutation.*
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§5 Element creation and transmutation

The coercion operator `!` is used to construct an element of a structure, or to map it into a structure, where a nature mapping exists.

```
> QQ := RationalField();
> QQ!17;
17
> P<x> := PolynomialRing(QQ);
> P![2,-2,1];
x^2 - 2*x + 2
```

Since a polynomial ring $R[x]$ is canonically defined by its base ring, elements can also be defined directly:

```
> Polynomial([2,-2,1]);
x^2 - 2*x + 2
```

Other `Magma` objects are created almost exclusively by creating a parent structure and using the `!` operator.

§5 Element creation and transmutation [cont]

Remember that the parent of a polynomial determines the interpretation of many functions which operate on it:

```
> K<i> := QuadraticField(-1);
> PK<x> := PolynomialRing(K);
> Factorization(Polynomial([2,-2,1]));
[
    <x^2 - 2*x + 2, 1>
]
> Factorization(Polynomial([K|2,-2,1]));
[
    <x - i - 1, 1>,
    <x + i - 1, 1>
]
```

In this case the sequence `Universe` determines the base ring of the parent polynomial ring.

§5 Element creation and transmutation [cont]

Automatic coercion occurs systematically throughout `Magma`. Consider the following examples:

```
> f := hom< QQ -> QQ | x :-> x >;
> f(2);
2
```

In this example, the input *integer* must be coerced into the domain (the *field of rationals*).

Now consider what must happen in this call to `eq`:

```
> 1 eq 15/77;
false
> FiniteField(2)!1 eq 15/77;
true
```

A common superstructure, either $\mathbb{Q}$ or $\mathbb{F}_2$, is found where 17 and 17/1 can be compared, and both elements are coerced into this structure.

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. *New structures from old.*
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§6 New structures from old

The construction of objects in Magma is recursive, we can create a rational function field F over a finite field, and create a quaternion algebra B over this function field.

```
> K<u> := FiniteField(3);
> F<x> := FunctionField(K);
> B<i,j,k> := QuaternionAlgebra< F | -1, x >;
> B;
Quaternion Algebra with base ring Univariate rational function field over GF(3)
> [ x*y : x, y in [i,j,k] ];
[ 2, k, 2*j, 2*k, x, 2*x*i, j, x*i, x ]
```

We can then form a sequence of products of elements in the algebra B.

```
> [ x*y : x, y in [i,j,k] ];
[ 2, k, 2*j, 2*k, x, 2*x*i, j, x*i, x ]
```

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. *Built-in operators.*
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§7 **Built-in operators**

We've already seen the assignment `:=` and coercion `!` operators.

Eltseq. In many instances, the coercion operator `!` can accept a defining sequence for an object. In such circumstances, the definition of `ElementToSequence` (or its shorthand `Eltseq`) should be such that `!` is an inverse operation.

Arithmetic operations. The standard arithmetic operators `+, -, *, /, ^` are defined for many categories. Where they exist, the standard assignment versions also exist `+:=, -:=, *:=, /:=, ^:=`.

N.B. In noncommutative rings, like matrix algebras, or in nonabelian groups or semigroups, the assignment operator `*:=` is a right multiplication assignment; no syntax exists for left multiplication assignment operator presently exists.

§7 Built-in operators [cont]

Integral division and remainder. The operators `mod` and `div` are defined such that `n` equals `(n div m)*m + (n mod m)` and `n mod m` is a nonegative number at less than the absolute value of `m`.

Boolean operators. The unary operator `not` and the binary operators `and` and `or` operate on the booleans `true` and `false`.

Comparison operators. The operator `eq` tests for equality of objects in `Magma`, returning a boolean, and for objects which have a ordering or partial ordering, the comparison operators are `le`, `lt`, `gt`, and `ge`.

Sequence and string operators. Strings and sequences are elements of free monoids for which `cat` or `*` serve as the binary operation.

Set operators. Sets admit the operators `join` and `meet`, as well as boolean operators `subset` and `in`.

§7 **Built-in operators [cont]**

Recursion on operators Any of the above binary operators, say `op`, which satifies an associative law gives rise to a recursive operator `&op` which applies to `sequences`. If the operation is also commutative, then a recursion operator applies to `sets`.

```
> s := &*[ "I", "n", "t", "e", "g", "e", "r" ];
> t := &*[ "R", "i", "n", "g" ];
> s cat " " cat t;
Integer Ring
```

§7 **Built-in operators [cont]**

N.B. There are no functions `Sum` or `Product` in `Magma`, because the recursion operators `&+` and `&*` fill these voids. The recursion operators `&op` can be very useful, as demonstrated by this one line implementation of the `subset` operator.

```
> X := {1..100};
> Y := { a : a in X | IsOdd(a) };
> &and[ a in X : a in Y ];
true
> Y subset X;
true
```

Membership and enumeration operators. The operator `in` is overloaded as both an membership operator and as an enumeration operator, as demonstrated in the above example.

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. *Language syntax.*
9. Functions and procedures.
10. Packages and intrinsics.
11. An example of quaternions.

§8 Language syntax

A. Language conventions. Functions in `Magma` are upper case and *should* refer to the noun which they return. For example, instead of the verb `Factor`, `Magma` uses the noun form:

```
> Factorization(2^(2^7)+1);
[ <59649589127497217, 1>, <5704689200685129054721, 1> ]
```

Syntax bugs. There exist exeptions to this convention, e.g. there exists a function named `Evaluate` rather than `Evaluation`.

B. Loops and flow control. The most commonly used flow control routines are `if`, `for`, and `while` loops.

```
if P in S then        while P in S do       for P in S do
   ...;                  ...;                  ...;
end if;               end while;            end for;
```

The `if` statement also permits `elif..then` and `else` clauses. Note the two distinct `in` operators in the `for`, `if`, and `while` routines.

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. *Functions and procedures.*
10. Packages and intrinsics.
11. An example of quaternions.

§9 Functions and procedures

Consider the file `my_function.m` with content:

```
function X(A,B)
   A +:= B;
   return A;
end function;
```

and the file `my_procedure.m` with content:

```
procedure X(~A,B)
   A +:= B;
end procedure;
```

Back in the `Magma` shell we `load` and use these functions.

```
> load "my_function.m";
Loading "my_function.m"
> A := 2; B := 7;
> X(A,B);
9
```

§9 **Functions and procedures [cont]**

But notice that the global variable `A` remains unchanged by the function.

```
> A;
2
```

In contrast the variable `A` is passed by reference, with `~A`, to the procedure `X` and can be changed.

```
> load "my_procedure.m";
Loading "my_procedure.m"
> X(~A,B);
> A;
9
```

`Magma` **`functions`** and **`procedures`** have no type checking of arguments, and overwrite any and all functions or **`intrinsics`** of the same name.

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. *Packages and intrinsics.*
11. An example of quaternions.

§10 **Packages and intrinsics**

`Intrinsics` include all functions or procedures with type checking and overloading which are built into the kernel of `Magma` (written and compiled in C). It is possible to view the `signature` of any such function from the `Magma` shell. E.g.

```
> HyperellipticCurve;
Intrinsic 'HyperellipticCurve'

Signatures:

    (<RngUPolElt> f, <RngUPolElt> h) -> CrvHyp

        Returns the hyperelliptic curve defined by the
        equation y^2 + h(x) y = f(x).
```

(followed by many more signatures for the same function)

§10 **Packages and intrinsics [cont]**

More and more `intrinsics` are being written in the `Magma` language, as part of `packages` distributed with the system. All such `Magma` code is in human readable form in the various subdirectories of

`$MAGMA_ROOT/package/`,

where `$MAGMA_ROOT` is the root directory where `Magma` is installed.

```
> ls
Aggregate  Geometry   Lattice  RepThry    spec
Algebra    Group      LieThry  Ring
Code       HomAlg     Module   Semigroup
Commut     Incidence  Opt      System
```

Additional source code for arithmetic geometry is available as share packages from from my web page

`http://magma.maths.usyd.edu.au/~kohel/magma/`,

§10 **Packages and intrinsics [cont]**

Consider the file `my_intrinsic.m` with content:

```
intrinsic X(A::RngIntElt,B::RngIntElt) -> RngIntElt
   {Returns the sum of A and B.}
   A +:= B;
   return A;
end intrinsic;

intrinsic X(~A::RngIntElt,B::RngIntElt)
   {Assigns the sum of A and B to A.}
   A +:= B;
end intrinsic;
```

The file `intrinsic.m` constitutes an integer addition package. Two `intrinsics` are defined, one is a function `X` and the second a procedure `X`. We use the package by means of the `Attach` command.

§10 **Packages and intrinsics [cont]**

```
> Attach("my_intrinsic.m");
> A := 2; B := 7;
> X(A,B);
9
> X(~A,B);
> A;
9
```

In a Unix shell, we also notice that magma has created a new file, called `my_intrinsic.sig` file (and in V2.11 and prior, a second file called `my_intrinsic.dat`).

```
chipotle ~> ls my_intrinsic*
my_intrinsic.dat  my_intrinsic.m  my_intrinsic.sig
```

The former is the compiled file, and the latter is a signature, which is checked at each carriage return in the `Magma` shell, to see if the file has changed and needs to be recompiled.

§10 **Packages and intrinsics [cont]**

```
> Attach("my_intrinsic.m");
> X;
Intrinsic 'X'

Signatures:

(<RngIntElt> A, <RngIntElt> B) -> RngIntElt

Returns the sum of A and B.

(<RngIntElt> ~A, <RngIntElt> B)

Assigns the sum of A and B to A.
```

Magma and Applications

1. The Magma shell.
2. Parents and categories.
3. Primitive structures.
4. Aggregate structures.
5. Element creation and transmutation.
6. New structures from old.
7. Built-in operators.
8. Language syntax.
9. Functions and procedures.
10. Packages and intrinsics.
11. *An example of quaternions.*

§11 **An example of quaternions**

We construct the order $\mathcal{O}$ of level 7 in the quaternion algebra B and enumerate representatives of its ideal classes.

```
> O := QuaternionOrder(19,7);
> time S := LeftIdealClasses(O); S;
[
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring,
    Quaternion Order of level (19, 7) with base ring Integer Ring,
    Quaternion Ideal of level (19, 7) with base ring Integer Ring
]
Time: 0.680
```

§11 An example of quaternions

Elements of the orders in a quaternion algebra B print with respect to their embedding in the algebra.

```
> B := QuaternionAlgebra(O);
> Basis(B);
[ 1, i, j, k ]
> Basis(MaximalOrder(B));
[ 1, i, j, k ]
> Basis(O);
[ 1, -2*i + j, 1 - i - j - k, -2*i - j + k ]
```

In particular, the right orders, which are also orders of level 7, will not generally have integral coordinates with respect to the embedding in B.

```
> Basis(RightOrder(S[1]));
[ 1, -1/4 - 3/4*i - 1/2*j, -i + 2*k, 3/4 + 21/4*i - 3/2*j - k ]
```

§11 An example of quaternions

We can also determine Minkowski reduced representatives of left or right ideal classes.

```
> [ Norm(I) : I in S ];
[ 4, 2, 4, 4, 8, 8, 2, 2, 4, 4, 1, 4 ]
```

```
> [ Norm(ReducedLeftIdealClass(I)) : I in S ];
[ 4, 2, 4, 4, 5, 5, 2, 2, 3, 3, 1, 3 ]
```