An Introduction to Magma

Don Taylor

The University of Sydney

13 February 2012

A little history

MAGMA is a programming language for computer algebra, geometry, combinatorics and number theory. It has extensive support for group theoretic computations and can handle permutation groups, matrix groups and finitely-presented groups. A more complete list of applications will be given later.

The language was developed by John Cannon and his team at the University of Sydney and was released in December 1993. It replaced CAYLEY, also developed by John Cannon.

Magma 2.18

The name 'Magma' comes from Bourbaki (and Serre) where it is used to denote a set with one or more binary operations without any additional axioms. Thus a *magma* is the most basic of algebraic structures.

Today, in 2012, Version 2.18 of Magma is a huge system with more than 5000 pages of documentation in 13 volumes.

Magma can be used both interactively and as a programming language. The core of Magma is programmed in C but a large part of its functionality resides in *package files* written in the Magma user language.

Outline

The following topics will be illustrated by examples drawn from a selection of MAGMA packages.

1. Using MAGMA interactively
   - Using files.
   - Printing.
   - The help system.
2. Essential data structures: sets, sequences, records and associative arrays.
3. Functions and procedures, function expressions and maps.
4. MAGMA's type system and coercion.
5. The MAGMA language and styles of programming.

The Magma design

The design principles underpinning both the user language and system architecture are based on ideas from universal algebra and category theory.

The Magma language attempts to approximate the usual mathematical modes of thought and notation as closely as possible. In particular, the principal constructs in the user language are sets, algebraic structures such as groups, rings and fields and morphisms.

[The following three slides are from the Magma documentation.]

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma language

- Imperative language with standard imperative-style statements and procedures
- A functional subset providing closures, higher-order functions, and partial evaluation
- General aggregate data types based on algebraic notions: set, sequence, mapping, magma
- Universal structure constructors providing a general mechanism for the construction of magmas and mappings
- Simple but powerful notation for constructing sets and sequences in a natural mathematical style
- Set and sequence operations which are implemented with a strong emphasis on efficiency
- Coercion between magmas (including automatic coercion)
- A package mechanism to support modular program construction

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The Magma environment

- Command completion and interactive line editing
- History system with recall and editing of previous lines
- A hierarchical online help facility
- Packages containing user-defined intrinsics with automatic compilation
- Environment variables for configuring style of output, etc.
- Get/set functions and procedures for configuring style, etc.
- Verbose options for built-in functions
- Logging of output, redirection of I/O
- Special file type for fully-featured file I/O
- Ability to execute system commands from within Magma
- Input/output pipes for communication with external programs

The main components of Magma

- The Magma Language and System
- Groups
- Semigroups and Monoids
- Rings and Fields
- Commutative Rings
- Linear Algebra and Module Theory
- Lattices and Quadratic Forms
- Algebras
- Representation Theory
- Homological Algebra
- Lie Theory
- Algebraic Geometry
- Finite Incidence Geometry
- Differential Galois Theory
- Error-correcting Codes
- Cryptography
- Mathematical Databases

Interacting with Magma

To start, simply type

```
magma
```

To stop, type `Ctrl-D` or

```
> quit;
```

Every instruction to Magma must end with a semicolon (;) but don't type the > sign—this is Magma's prompt character.

When developing Magma programs it is useful to put the commands in a file. To load the commands from a file named `lecture1.m`, type

```
> load "lecture1.m";
```

If you would like Magma to pause after each line of input, use

```
> iload "lecture1.m";
```

Using MAGMA interactively — Tab completion

Suppose that you want to find the first few Legendre polynomials. There are many ways to achieve this in MAGMA.

One of the first things to do is to check the documentation to find out if there is already a function which does what you want.

Alternatively you can use the MAGMA help system. For example, if you type `Legendre` at a MAGMA prompt and then press the `Tab` key you will see the following:

```
> Legendre
LegendreEquation        LegendrePolynomial
LegendreModel           LegendreSymbol
```

If you now type `P` followed by `Tab` and a semicolon:

```
> LegendrePolynomial;
```

you will see some information about how to use this function.

Using MAGMA interactively — the Help system

To get more detailed information, put a question mark before the name of the command:

```
> ?LegendrePolynomial
```

Depending on how MAGMA is installed either a web browser will open showing the online help for the function or MAGMA's internal help system will provide information about the command.

To print the 7th Legendre polynomial simply issue the command

```
> LegendrePolynomial(7);
```

The output is

```
429/16*$.1^7 - 693/16*$.1^5 + 315/16*$.1^3 - 35/16*$.1
```

To get more readable output it is possible to assign a more useful name, such as z, to the indeterminate `$.1`. We shall see how to do this on the next slide.

Legendre Polynomials: Rodrigues' formula

Suppose that you didn't discover that MAGMA has a built-in function for Legendre polynomials. In that case you can define them yourself using either a direct or recursive formula.

For the direct method we use Rodrigue's formula

$$P_n(z) = \frac{1}{2^n\, n!} \frac{d^n}{dz^n}(z^2 - 1)^n.$$

To set this up in MAGMA we need the polynomial ring in the indeterminate z over the rational numbers.

```
> R<z> := PolynomialRing(Rationals());
> P := func< n | Derivative((z^2-1)^n,n)/(2^n*Factorial(n))>;
> P(7);
429/16*z^7 - 693/16*z^5 + 315/16*z^3 - 35/16*z

> P(7) eq LegendrePolynomial(7);
true
```

Legendre Polynomials: a recurrence relation

Another way to define the Legendre polynomials is via the recurrence relation

$$nP_n(z) = (2n-1)zP_{n-1}(z) + (n-1)P_{n-2}(z), \quad P_0(z) = 1, \quad P_1(z) = z.$$

Using this definition in MAGMA illustrates

- assignment `:=`
- sequences `[ <expression> : n in range ]`
- the `select` expression
- recursive definitions using `Self`
- the range expression: `[a..b by c]`

```
> L := [ n eq 0 select 1 else n eq 1 select z else
>   ((2*n-1)*z*Self(n)-(n-1)*Self(n-1))/n : n in [0..7]];
> L[8];
429/16*z^7 - 693/16*z^5 + 315/16*z^3 - 35/16*z
```

Note that MAGMA sequences are indexed from 1, not 0.

Number systems

The number systems in MAGMA go well beyond the usual provision of integers, rationals and real numbers. MAGMA has extensive support for many types of rings, finite fields, and local and global number fields.

To illustrate this, construct the cyclotomic field $E = \mathbb{Q}[\omega]$, where $\omega^3 = 1$ and then define three matrices over E.

```
> E<w> := CyclotomicField(3);
> E;
Cyclotomic Field of order 3 and degree 2
> a := Matrix(2,2,[E| w, 0, w^2, 1]);
> b := Matrix(2,2,[E| 1, -w^2, 0, w]);
> c := Matrix(2,2,[E| 2*w+1, 2*w, w^2, -w]);
```

It turns out that the group generated by a and b has order 24, but the group generated by a and c is infinite.

```
> G1 := sub<GL(2,E) | a, b >;
> G2 := sub<GL(2,E) | a, c >;
> #G1, IsFinite(G2);
24 false
```

Strings

Strings are enclosed in double quotes and the backslash character is used as an escape. That is, to obtain a double quote, a backslash, a newline or tab, you would type `\"`, `\\`, `\n` or `\t`.

Use `IntegerToString` to convert an integer to a string.

The expression `s * t` is the concatenation of strings `s` and `t`.

```
> base := "E";
> for n := 6 to 8 do // or you could use n in [6,7,8]
for>   print base*IntegerToString(n);
for> end for;
E6
E7
E8
```

As in languages such as C++ and Java, `//` introduces a comment which extends to the end of the line and `/* comment text */` is a comment which may span several lines.

Boolean values and relational operators

MAGMA has the usual two Boolean constants `true` and `false` and the usual connectives `and`, `or`, `xor` and `not`.

But unlike many other programming languages MAGMA uses abbreviations rather than symbols for its relational operators. For example, to determine whether `x` is less than or equal to `y` you type `x le y` not `x <= y`.

MAGMA code	meaning
`x eq y`	$x = y$
`x ne y`	$x \neq y$
`x lt y`	$x < y$
`x le y`	$x \leq y$
`x gt y`	$x > y$
`x ge y`	$x \geq y$
`x in A`	$x \in A$

Data structures: sequences and tuples

Only elements of a common structure are allowed in sets or sequences. The common structure is its *universe*.

Sequences

```
> a := [2,3,5,7,11];
> b := [ x^2 : x in a ];
> T := [ x : x in Sym(5) | Order(x) eq 2 ];
> Universe(a);
Integer Ring
```

A *tuple* is an element of a Cartesian product.

Tuples

```
> X := [ < x , Order(x) > : x in Sym(3) ];
> Universe(X):Minimal;
Cartesian Product<GrpPerm: $, Degree 3, Order 2 * 3, Integer Ring>
```

Data structures: sets and indexed sets

As in mathematics, *sets* do not contain duplicate elements and there is no preferred order of the elements.

Sets

```
> V := VectorSpace(GaloisField(4),4);
> G := SpecialUnitaryGroup(4,2);
> e := { V.1^g : g in G };
> #e;
135
```

An *indexed set* is a set whose elements are linearly ordered.

Indexed Sets

```
> A := {@ 3,3,2,1,5,3,5 @};
> print A;
{@ 3,2,1,5 @}
> print A[3];
1
```

Data structures: lists

Lists are ordered collections of objects of *any* type; that is, a *list* does not require its members to belong to a common universe.

Lists

```
> L := [* 3, G.1 *] where G is SymmetricGroup(4);
> L;
[* 3,
    (1, 2, 3, 4),
*]
> Append(~L, "symmetric group");
> L;
[* 3,
    (1, 2, 3, 4),
    symmetric group
*]
```

If `L` is a list or a sequence, `Append(~L,x)` adds the element `x` to the end of `L`, modifying `L` in place. The command `M := Append(L,x)` leaves `L` unchanged.

Data structures: records

Records are a useful and flexible way to store the results of a computation in fields indexed by field names.

For example, in carrying out a depth-first search to find a directed spanning forest of a digraph, you might encode the current state of the search as a record which is updated each time your procedure visits a node.

Records

```
context := recformat< forest, rho, counter >;

state := rec< context |           // n is the number of nodes
  forest  := [{Integers()|} : i in [1..n]],
  rho     := [0 : i in [1..n]],  // rank of the current node
  counter := 0                   // number of nodes processed
  >;
```

At the end of the computation `state‘forest` should hold a sequence of directed trees.

Data structures: attributes

An *attribute* of a category (such as the category `GrpMat` of matrix groups) is a value which can be attached to an instance of the category and accessed via named fields in the way that the fields of a record are accessed.

```
> G := CoxeterGroup(GrpMat,"H4");
> G`Order;
14400
```

You can define your own attributes using the command `AddAttribute(C,F)`, where `C` is a category and `F` is the name of the attribute (as a string).

An example will be given on the last slide.

Data structures: associative arrays

An *associative array* is a sequence which can be indexed by arbitrary members of a given universe. In other programming languages this data structure is called a *dictionary*, *table* or *hash map*.

Associative arrays

```
> A := AssociativeArray();
> for n in [6,7,8] do
>   name := "E"*IntegerToString(n);
>   A[name] := CoxeterGroup(name);
> end for;
> Order(A["E6"]);
51840
```

Functions and procedures

In MAGMA there is distinction between *functions* and *procedures*: a function returns (possibly several) values; a procedure does not return a value.

```
> BinaryOctahedral := function()
>   P<x> := PolynomialRing(IntegerRing());
>   K<i,q> := NumberField([x^2+1,x^2-2]);
>   a := [g/2 : g in [-i-1, -i+1, -i-1, i-1]];
>   c := [(1-i)/q,0,0,(1+i)/q];
>   return sub< GL(2,K) | a, c >;
> end function;
```

A procedure can modify an argument, such as `A` provided it occurs in the argument list as a *reference variable*, written `~A`.

The following procedure adds m times row i to row j of the matrix A.

```
> addrow := procedure( ~A, i, j, m )
>   n := Ncols(A);
>   for k := 1 to n do  A[j,k] +:= m*A[i,k]; end for;
> end procedure;
```

Maps

When MAGMA constructs a *subgroup* of a group it also returns a *map* from the subgroup to the group.

```
> G := Sym(5);
> H,f := sub< G |  (1,2)(3,4), (1,2,3,4) >;
> f;
Mapping from: GrpPerm: H to GrpPerm: G
```

There are several ways to create maps. Here is one way to create the Frobenius homomorphism of a finite field.

```
> q := 3;
> F<iota> := GF(q^2);
> sigma := map< F -> F | x :-> x^q >;
> sigma(iota+2);
iota^5
```

Control structures

while

```
while boolean do statements; end while;
```

repeat-until

```
repeat statements; until boolean;
```

for

```
for i := a to b by c do statements; end for;
for i in S do statements; end for;
```

if-then-else

```
if boolean then statements;
elif boolean then statements;
else statements;
end if;
```

Other useful statements and expressions

- `case` expressions
- `case` statements
- `select` statements
- `error` and `error if` statements
- `try` — `catch e` — `end try`
- `continue`
- `break`
- `where ... is`
- `exists`
- `forall`

Magma's type system

(Almost) every object in Magma belongs to a *category*, also known as the *type* of the object. In addition, every object has a *parent*.

```
> G := Alt(4); // the alternating group on {1,2,3,4}
> G;
Permutation group G acting on a set of cardinality 4
Order = 12 = 2^2 * 3
    (1, 2)(3, 4)
    (1, 2, 3)
> Type(G);
GrpPerm
> Parent(G);
Power Structure of GrpPerm
> Generic(G);
Symmetric group acting on a set of cardinality 4
Order = 24 = 2^3 * 3
```

Coercion

Suppose that V is a vector space of dimension 3 over the rational numbers. In MAGMA the elements of V are triples of rational numbers; i.e., row vectors. However, a triple `[2,3,7]` represented as a sequence will not be recognised as an element of V.

```
> V := VectorSpace(Rationals(),3);
> v := [2,3,7];
> v in V;

>> v in V;
      ^
Runtime error in 'in': Bad argument types
```

In order to have MAGMA recognise v as an element of V it must be *coerced* into V.

```
> vec := V!v;
> vec in V;
true
```

Styles of programming

MAGMA is an imperative, call by value, lexically scoped, dynamically typed programming language, with an essentially functional subset.

An *imperative language* manipulates data directly by assignment statements and control structures such as loops and if-else constructions. You can do this in MAGMA.

A *functional programming language* achieves its goals by composing functions without side-effects. In MAGMA you can do this too.

Functions are a fundamental part of MAGMA programming and they are *first-class*. That is, they can be assigned to variables, stored in data structures and returned from other functions. They can invoke themselves recursively and participate in mutual recursion.

MAGMA also has procedures and unlike a function a procedure does not return a value but it may modify its arguments (provided they are declared as *reference* variables).

Styles of programming — closures

A *closure* is a function that has access to local variables from a larger scope, namely the context in which it is defined.

```
> add_three := function()
>   y := 3;
>   return func< x | x + y >;
> end function;
> y := 7;
> add_three()(10);
13
> add_factory := function(y)
>   return func< x | x + y >;
> end function;
>
> add3 := add_factory(3);
> add3(10);
13
```

Final example: efficient recursion

The Chebyshev polynomials $T_n(z)$ (of the first kind) may be defined by the recursion

$$T_n(z) = 2zT_{n-1}(z) - T_{n-2}(z), \quad T_0(z) = 1, \quad T_1(z) = z$$

```
> chebyshev := function(n)
>   chebyshev_ := procedure(~Z,z,n)
>     if not IsDefined(Z`cheb_poly,n+1) then $$(~Z,z,n-1);
>       Append(~Z`cheb_poly, 2*z*Z`cheb_poly[n]-Z`cheb_poly[n-1]);
>     end if;
>   end procedure;
>
>   AddAttribute(Rng,"cheb_poly");
>   P<z> := PolynomialRing(Integers());
>   P`cheb_poly := [P| 1, z];
>   chebyshev_(~P,z,n);
>   return P`cheb_poly[n+1];
> end function;
> time _ := chebyshev(500);
Time: 0.030
```