Introduction to Magma

Lecture and Hands On Session at UNCG 2016

Florian Hess
University of Oldenburg

What is Magma?

Magma is a computer algebra system for computations in algebra and geometry.

- Has an interpreter which allows for interactive computations,
- has an own purpose designed programming language,
- has an extensive help system and documentation,
- strives to provide a mathematically rigorous environment,
- strives to provide highly efficient algorithms and implementations.

Magma consists of a large C kernel to achieve efficiency, and an increasingly large package of library functions programmed in the Magma language for higher functionality.

The name derives from “magma” in the sense of Bourbaki, “a set with composition law”.

What is Magma?

Magma is developed by the Computational Algebra Group at University of Sydney, headed by John Cannon.

A large number of external collaborators has contributed significantly.

More information about current and former group members and external collaborators can be found under

`http://magma.maths.usyd.edu.au/`.

Early versions date back to the late eighties.

Has serveral million lines of C code, and several hundred thousand lines of Magma code.

A simplified online Magma is available under

`http://magma.maths.usyd.edu.au/calc`.

What is Magma?

Magma deals with a wide area of mathematics:

- Group theory
- Basic rings, linear algebra, modules
- Commutative algebra
- Algebras, representation theory, homological algebra, Lie theory
- Algebraic number theory
- Algebraic geometry
- Arithmetic geometry
- Coding theory, cryptography, finite incidence structures, optimisation

It is not useful for engineering maths (differentiating, integrating, ...), expression simplification, etc.

What is Magma?

Magma is an executable that comes with a collection of library and documentation files, residing all in a directory and subdirectories.

Magma is started by typing “magma”. After starting Magma, a user interactively enters lines of Magma language, thereby performing some calculations by hand, or defining and executing whole programmes.

The execution of programmes is terminated by Ctrl-C (may take some time). The execution of Magma is terminated by typing “quit;” or Ctrl-D at its prompt, or hitting Ctrl-C twice within half a second, or hitting Ctrl-\.

A first look (just starting, `1 + 1;` and quitting) ...

Documentation

Online documentation and help can be obtained as follows:

- typing “magmahelp” at a shell prompt, or google “magma online help” and hit link ...
- typing the name of a Magma command followed by “`;`” into Magma.
- Pressing the Tab-key twice.
- Using ? and ?? ...

Let’s do this ...

Design Criteria

The mathematical viewpoint of Magma is to emphasize algebraic and geometric structures (being defined by or satisfying certain axioms) together with the corresponding morphisms.

The Magma language is designed to suit this viewpoint well.

Specific criteria:

- Universality: Should be appropriate for a broad range of areas within algebra and geometry.
- Mathematical system:
  - Structures and morphisms should be “first class objects”.
  - Precise, unambigous and similar formulation of mathematical content by the language. Emphasis of “mathematical” data structures over “computer science” data structures.
- Efficiency.

Universality

General trouble of computer algebra:

- One system or concept for all of mathematics very difficult, probably impossible or insensible.
- Very specialised systems or concepts may soon need tools from neighbouring areas.

Magma hopes to be sufficiently general and sufficiently constrained to enable a large class of mathematical computations within the chosen mathematical viewpoint.

Mathematical system

- Structures (and morphisms) should be available for operations like “normal elements”.
- Must be possible to express mathematical content and context in a precise and unambigous way.
- Support sets, sequences, mappings and the respective mathematical operations.

Examples 1:

- Factorisation of $x^3 - 2$? Define correct polynomial ring, define x, then ask for factorisation.
- Define number field, ask for class number.
- Supports also various concurrent mathematical contexts.
- Use sets instead of removing double entries by for-loops over lists.

Efficiency

- Magma is intended as practical, heavy-duty research tool.
- Avoid use of generic data structures and implementation, use highly specialised data structures and implementation in C and link with general concept.
- Higher level data structures and algorithms may then be implemented in Magma language without loss of efficiency.

Examples 2:

- Integers, polynomials, matrices, etc.
- Reduction operator over sequences works in the C to avoid slow interpreter.

The Magma model

The Magma model, or mathematical viewpoint, is based on concepts from universal algebra ($\sum$-algebras) and category theory.

Some informal definitions:

- A variety is a class of objects satisfying common basic axioms.
- A category (or type) is a class of objects belonging to a variety that share the same "representation".
- An extended category is a class of objects belonging to a category and being parametrised by a category.
- Every object belongs to a (unique, extended) category.
- Every object has a (unique) parent structure, describing the mathematical context in which it is viewed.

We do not go into further details about this here.

The Magma model

Example:

- Rings form the variety `Rng`.
- Univariate and multivariate polynomial ring form the categories `RngUPol` and `RngMPol`.
- Univariate polynomials over the integers form the extended category `RngUPol[RngInt]`.
- The parent of a univariate polynomial is its polynomial ring.

Magma has various functions to retrieve and compare categories of objects.

- `ISA(),ListSignatures()`,...

Examples 2 ctd. ...

The Magma model

The creation of free objects, subobjects, quotient objects and extension objects are standard operations and are supported by providing special constructors for these operations.

Furthermore, there are constructors for element and map creation.

Maps can be partial and are generally represented by providing

- the graph of the map,
- a rule (also for the inverse),
- images in the codomain of the generators of the domain.

Coercion helps dealing with large numbers of types.

If there is a natural map of a structure A into a structure B, objects of A may be used where objects of B are expected, by invoking the map automatically. Examples 3 ...

The Magma language

The Magma language has the following features:

- imperative,
- call by value,
- dynamically typed,
- with an essentially functional subset.

Magma code can be typed directly into Magma or be read into Magma from ".m" files using the `load` statement.

In the following a quick overview over the Magma language.

Expressions and identifiers

An object in Magma is specified by an expression. An expression is composed of less complex expressions and identifiers.

There are three classes of identifiers:

- variable identifiers, which may be assigned values,
- value identifiers, are constants, may not be assigned values,
- reference identifiers, preceded by $\sim$, may be used to return values from procedures.

An identifier cannot belong to more than one of these classes.

Identifier names must begin with a letter and must be distinct from reserved words. They are case sensitive.

Assignments

There are various forms:

- `a := 1;`
- `a[2][3] := 1;` if `a` is indexed.
- `a, b := f(1);` if `f` has multiple return values.
- `_, b := f(1);` forget the first return value.
- `a op:= 1;` equivalent to `a := a op 1;`, more efficient.

- `R<x1, x2> := PolynomialRing(Integers(), 2);` assign ring and generators.
- `R<[x]> := PolynomialRing(Integers(), 2);` assign ring and sequence of generators.

- `assigned a;` boolean value whether `a` is defined.
- `delete a;` unassign `a` such that it becomes undefined.

Boolean values and comparisons

Boolean values are `true` and `false`.

- There are the usual boolean operators `and`, `or`, `xor`, `not`.
- The evaluation of boolean expressions is stopped as soon as result is known!

Comparisons yield boolean values or signal an error.

- `a ge b;` greater equal, error if incomparable.
- Similarly `gt`, `le`, `lt`, `eq`, `ne`.
- `a cmpeq b;` if incomparable return `false`, otherwise like `eq`.
- `a cmpne b;` if incomparable return `true`, otherwise like `ne`.

Conditionals

Conditional statement as usual (`elif` and `else` parts may be omitted):

```
if bool-expr-1 then
  statements-1
elif bool-expr-2 then
  statements-2
else
  statements-3
end if;
```

Conditional expression:

`c := bool-expr select a else b;` choose `a` or `b` depending on `bool-expr`.

There is also a `case` statement and expression.

Iterative statements

Usual loops

- `for i := expr-1 to expr-2 by expr-3 do statements end for;` (the `by` part may be omitted).
- `for i in S do statements end for;` where `S` allows iteration.
- `while bool-expr do statements end while;`
- `repeat statements until bool-expr;`
- Early exit with `break;` statement in the statements.
- Early exit in `for` loop over `i` can be specified by `break i;` in case of nested `for` loops.

Functions

General function definition:

```
f := function( x1, ..., xn : y1 := expr1, ... )
  local w1, ..., ws;
  statements
  return z1, ..., zr, _, ..., _;
end function;
```

The `x` are mandatory parameters and the `y` optional parameters taking the default values `expr`. The `z` are the return parameters. The `_` are undefined return values. The `w` are local variable identifiers.

Short form:

```
f := func< x1, ... : y1 := expr1, ... | expr >.
```

Functions

General calling of functions:

```
z1, ..., zm := f( x1, ..., xn : y1 := expr1, ... );
```

Any of the `y` may be omitted. Those who are assume the default value in the function body. Any of the `z` may be `_` in which case the corresponding result is thrown away.

The final `_` in the return statement may be included to provide a consistent number of return values (`b, c := f(a);` causes an error if not always two values are returned).

Example:

- `b, c := IsSquare(a);` returns a boolean `c` and a square root of `a` if it exists, and `_` otherwise.

Procedures

General procedure definition:

```
f := procedure([~]x1, ..., [~]xn : y1 := expr1, ...)
  local w1, ..., ws;
  statements
end procedure;
```

Procedures are like functions without return statements, but with reference identifiers `~x` which allow to return results through the arguments.

General calling of procedures:

```
f( [~]x1, ..., [~]xn : y1 := expr1, ... );
```

Functions and Procedures

Example:

```
> f := function(a : MyOpt := true ) return MyOpt;
function> end function;
> f(3);
true;
> f(3 : MyOpt := false);
false;
>
> h := procedure(~a) a := a + 1; end procedure;
> b := 1;
> h(~b);
> b;
2
```

Recursive functions and procedures

Reference to the function or procedure being defined from within its body via the identifier `$$` (the function or procedure identifier is not yet defined).

Alternatively, before the declaration of the function `f` use the statement `forward f;`. Then `f` can be accessed from within its body via the the identifier `f`.

An infinite recursion causes a segmentation fault.

Example:

```
f := func< i | i le 0 select 0 else i + $$(i-1) >;
```

```
forward f;
f := func< i | i le 0 select 0 else i + f(i-1) >;
```

Magma semantics

Within a function or procedure, the type of a variable, if not a reference variable or an argument, is determined by the first use rule.

- First textual use when it is assigned a value: Variable identifier, could have been declared by `local` statement (this is not necessary).
- First textual use when its value is accessed: Value identifier, must have been declared outside the function.
- Value computation in assignments comes first (`a := a;`).

Magma semantics

Objects are immutable (with exceptions!): The value of an identifier can change only by explicit reassignment using `:=` or ˜.

- Functions cannot change the identifiers of the calling context, procedures only through ˜.
- Once a function or procedure is defined, it does not change. This is useful for storing information within a function or procedure which might be necessary for a computation.

Example:

```
> a := 1;
> f := function(b) return b + a; end function;
> a := 2;
> f(1);
2
```

Comments, errors and assertions

Comments (like in C++):

- One line comment `// text`
- Enclosed comment `/* text */`

Error checking and assertions:

- `error expr-1, ..., expr-n;` print expressions and signal error (returns to read-eval loop).
- `error if bool-expr, expr-1, ..., expr-n;` only if condition is true.
- `assert bool-expr;` error if condition is false. Check may be switched off by `SetAssertions(false);`.

Used when programming.

Sets

The elements of a set must have the same parent (the universe of the set, obtained by `Universe()`).

There are various finite set types:

- Enumerated sets `{ ... }`,
- Indexed sets `{@ ... @}` (allows indexing),
- Multisets `{* ... *}` (elements occurs with multiplicity),

Creation (using the desired brackets):

- `{ U | x1, ..., xn }`
- `{ U | expr : x1 in E1,..., xn in En | bool-expr }`

`U` denotes the resulting universe. The `U |` or `| bool-expr` parts may be omitted. The set type is indicated by the brackets.

Sets

Creation:

- `{ x1 .. xn }`, `{ x1 .. xn by d }` for integer values.
- `Set(S)` for finite structure `S`.

Some operations:

- `#, eq, ne, in, notin, subset, notsubset, join, meet, diff`, ...
- `&+, &and, &meet, forall, exists`, ...
- `Random(), Representative(), ChangeUniverse(), Include(), Exclude()`, ...
- Conversion functions between the different types.

Sets

Examples:

- `{ 1, 2, 4 },{ 1 .. 100 },{ 1 .. 100 by 3 }`
- `S := {* 2^^3, 1^^2 *}; Multiplicity(S, 1);`
- `{@ Rationals() | x^2 : x in { 1 .. 100 } | IsOdd(x) @}`
- `{ }, { Integers() | }`
- `Set( GF(5) )`
- `{ 1, 2, 3 } meet { 2, 3, 4 }`
- `&+ { 1, 2, 3 }, &+ { Integers() | }, &and { true, true, false }`

Sequences

Quite analogous to finite sets. Brackets used are `[ ... ]`.

Operations:

- `cat, &cat, &meet,` ...
- `Random(), Representative(), Universe(), ChangeUniverse(), Reverse(), Position(), Append(), Insert(), Exclude(), Remove(), Sort(), And(),` ...
- Conversion to and from enumerated sets.

Example:

- Fibonacci numbers `[ i gt 2 select Self(i-2)+Self(i-1) else 1 : i in [1..100] ];`

Tuples and Cartesian Products

Tuples are elements of cartesian products. Cartesian products may be formed of any structures.

Creation of tuples:

- `< x1, ..., xn >`, `Append()`, `Prune()`, `Flat()`, ...

Access:

- `#T`, `T[i]`, ...

Creation of cartesian products:

- `CartesianProduct(S1, S2)`, `CartesianProduct( [ S1, ..., Sn ] )`, `CartesianPower(S, n)`, ...

Operations:

- `#`, `Component()`, `Representative()`, `Random()`, ...

Records

Finite collection of objects within one object, each of which is accessible by its own identifier (like usual).

To define a record, one first needs to define its record format. The format specifies the names of the components and possibly their types.

Creation by way of example:

- `RF := recformat< n : Integers(), misc, seq : SeqEnum >;`

Access:

- `Names(RF);`

Records

Creation of a record:

- `s := rec< RF | misc := "dfdf", n := 42 >;`
- `r := rec< RF | >;`

Operations:

- `r`misc; ,r`seq := [ 1, 2 ];`
- `delete r`misc;,assigned r`misc;` (boolean).
- `Names(),Format(),`
- `r``s;` where `s` evaluates to a string.

Because of their flexible nature, there are no comparisons of records!

Attributes

It is possible to add components to structures which are then called attributes and are accessed like record components. This is used mostly in packages (e.g. to store computed structure invariants).

Example:

- `if assigned Z`misc then ...,Z`misc := 1; delete Z`misc;`
- `GetAttributes(RngInt);`

Also `declare attributes type : field-1, ..., field-n;` (in packages).

Maps

Maps are an independent data type in Magma (category `Map`), as opposed to functions (category `UserProgram`).

Creation of maps:

- When doing `sub< ... >, quo< ... >`.
- Via the graph: `map< A->B |[ <x1,y1>,...,<xn,yn> ] >`
- Via a rule: `map< A->B | x :-> f(x), y :-> g(y) >`
  `y :-> g(y)` may be omitted. Also `hom< ... >` possible.
- Via images on the generators:
  `hom< A->B | [ y1, ..., yn ] >`
  For rings: `hom< A->B | c, [ y1, ..., yn ] >`, `c` maps `BaseRing(A)` to `B`.

Partial maps (`pmap< ... >`) by graph or rule are also possible.

Maps

There is not much checking whether the definition of a map makes sense. The non generic operations below do not work for all structures.

Operations:

- Composition `f*g` (formal), `Components()`.
- `Domain(), Codomain(), Kernel(), Image(), Inverse()`.
- `f(a), a@f, a@@f, HasPreimage(a, f)`.
- Can be applied (image, preimage) elementwise on sets, sequences, ...
- Coercion maps corresponding to internal coercion functions.

Maps

Example:

```
> Zxy := PolynomialRing(PolynomialRing(Integers()));
> Zx := BaseRing(Zxy);
> c := hom< Zx -> Zx | x :-> Evaluate(x, 2) >;
> f := hom< Zxy -> Zx | c, Zx.1 >;
> f(Zxy.1);
Zx.1
> f(Zxy!Zx.1);
2
```

Input and Output

There are many functions dealing with input and output.

- `print expr-1, ..., expr-n;`
- `printf format, expr-1, ..., expr-n;`
  - e.g. `printf "One %o Two %o \n\n", 1, 2;`
- Print levels: `Default`, `Minimal`, `Maximal`, `Magma`.
  - e.g. `print expr : Maximal .`

Into files:

- `Write( F, x );` Append `x` to file with name `F`.
  - `Write( F, x : Overwrite := true );` overwrites.
- `fprintf file, format, expr-1, ..., expr-n;`

Printing into strings: With `Sprint(), Sprintf()`.

Input and Output

Further input and output operations:

- Redirecting output.
- C library style file operations.
- Pipes, sockets.
- Saving and restoring the complete workspace.
- System calls (some), executing external commands, creating temporary unique files names.

- Loading program files (`load, iload`).
- Logging a session (`SetLogFile(), UnsetLogFile(), SetEchoInput()`.

Packages and intrinsics

Serious programming will lead to a package for performing a collection of mathematical computations.

A package consists of several ".m" files, each of which defines functions, procedures and intrinsics.

Intrinsics are like functions or procedures plus

- required input types and output types are specified (the signature), thus overloading is possible (same name, but different functions).
- A short documentation can be stored with the intrinsic.

Defining an intrinsic in a package requires a special syntax (specifying types, comment. See statement `intrinsic`).

Functions and procedures defined in a package are not visible outside the package (unless imported using `import`).

Packages and intrinsics

There are two types of intrinsics:

- Intrinsics defined in a package are called user intrinsics.
- Built-in intrinsics are called system intrinsics, and correspond to functions in the C kernel.

Intrinsics are stored in a global table which can be inspected (e.g. by typing the intrinsic name, "`;`" and enter).

A package file can be imported using `Attach()` (not `load`).

A collection of packages to be imported can be specified by a spec file.

Packages and intrinsics

Packages are precompiled for faster loading.

- yields ".sig" and ".dat" files.
- temporary lock files ".lck" during compilation.
- Changes in a ".m" file are automatically detected and the corresponding ".sig" and ".dat" are updated.
- This does not happen when `freeze;` is given in the beginning of the ".m" file.

User Defined Types

Only in packages!

Declare new data type:

- `declare type T;`
- `declare type T: P1, ..., Pn;`
- `declare type T[E];`
- `declare type T[E]: P1, ..., Pn;`

Then `T` is the type of a new structure, `E` the type of its elements if any. `T` is inherited from `P1,...,Pn`, so `ISA(T, Pi)` is true.

Creating an object of type `T`:

- `New(T);`

User Defined Types

Required special intrinsics:

```
intrinsic Print(X::T)
{Print X}
    // Code: Print X with no new line, via printf
end intrinsic;

intrinsic Print(X::T, L::MonStgElt)
{Print X at level L}
    // Code: Print X at level L with no new line,
    //   via printf
end intrinsic;
```

`L` can be the strings `Minimal, Maximal, Magma`.

User Defined Types

Required special intrinsics for element types:

```
intrinsic Parent(X::T) -> .
{Parent of X}
    // Code: Return the parent of X
end intrinsic;
```

```
intrinsic 'in'(e::., X::T) -> BoolElt
{Return whether e is in X}
    // Code: Return whether e is in X
end intrinsic;
```

User Defined Types

Required special intrinsics for element types (ctd):

```
intrinsic IsCoercible(X::T, y::.) -> BoolElt, .
{Return whether y is coercible into X and the
 result if so}
    // Code: do tests on the type of y to see
    //   whether coercible
    // On failure, do:
    //    return false, "Illegal coercion";
    //   Or more particular message
    // Assumed coercible now; set x to result of
    //   coercion into X
    return true, x;
end intrinsic;
```

User Defined Types

Have unfortunately various shortcomings:

- Not possible to truly inherit from existing internal data types, including methods.
- For example, if a new ring type is to be defined it will not be possible to use the existing polynomials or matrices ...

Some useful extras

Here are some additional useful remarks.

- Use `.magmarc` file for predefining things (do not read: `magma -n`)
- External help browser:
```
SetHelpUseExternalBrowser(true);
SetHelpExternalBrowser("firefox %s /dev/null
&");
```
- Last results are accessed by `$1, $2, $3`. Increased by `SetPreviousSize()`
- When working with copy paste: `SetIgnorePrompt(true);`
- Don’t forget the profiler ...

Some useful extras

- Timing: `time statement`, `Cputime()`.
- Random elements of structures: `Random()`
- Reproducing results of random computations: `SetSeed()`, `magma -S`.
- Verbose printing for inside information: `SetVerbose()`, `ListVerbose()`, `vprintf`, ...
  - For Kant: `SetKantLevel(3);`
    `SetKantVerbose("ORDER_CLASS_GROUP_CALC", 7);`
- User defined verbose flags: `declare verbose "xyz", 7;`, only in packages.

Some useful extras

Running big jobs:

- Bound memory usage: `SetMemoryLimit()`
- Bound time: `Alarm()`
- Run in background:
  - `nohup nice magma < infile >& outfile &`.
  - `tail -f outfile`, Ctrl-C exits.
  - or use screen (unix, read `man screen` or `byobu`, many text windows in one window) ...