

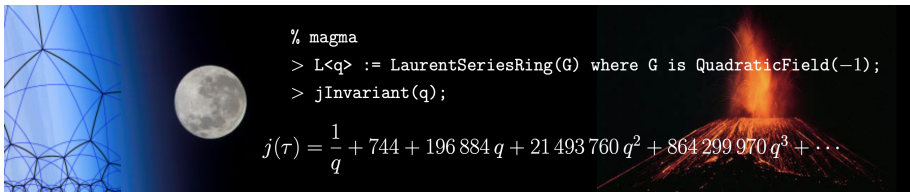
An Introduction to MAGMA

<https://www.maths.usyd.edu.au/u/don/presentations.html>

Don Taylor

The University of Sydney

9 October 2023



```
% magma  
> L<q> := LaurentSeriesRing(G) where G is QuadraticField(-1);  
> jInvariant(q);
```

$$j(\tau) = \frac{1}{q} + 744 + 196\,884q + 21\,493\,760q^2 + 864\,299\,970q^3 + \dots$$

Outline

- Day 1
 - MAGMA overview
 - The read-evaluate-print-loop (REPL)
 - Interactive programming
 - ▶ A simple word game
 - ▶ The Catalan numbers
 - ▶ Projective planes, graphs, automorphism groups
 - ▶ Exploring small groups: the Small Groups Database
- Day 2
 - The type system and coercion
 - Group theory examples
 - ▶ Constructing the Hall-Janko group
 - ▶ Group algebras and the group determinant
 - ▶ Central extensions of symmetric groups
- Day 3
 - Structure constant algebras
 - Lattices, root systems
 - Root data, reductive groups

A little history

MAGMA is a programming language for computer algebra, algebraic geometry, number theory and combinatorics.

It has extensive support for group theoretic computations and can handle permutation groups, matrix groups, finitely-presented groups, and groups of Lie type.

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.

The MAGMA language

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.

The MAGMA Handbook (in .pdf form) runs to more than 6000 pages and it is quite a daunting task to learn even a small portion of this material. <http://magma.maths.usyd.edu.au/magma/handbook/>

So, rather than attempting to cover every aspect of the language I shall begin with a simple example then describe the syntax and semantics as we explore various ways to solve and generalise the problem.

Before we begin

If MAGMA is installed on your computer, you will interact with it via the command line.

Typing `magma` will get you to the MAGMA prompt `>` and enter the 'read-evaluate-print-loop' (REPL).

To exit MAGMA type `quit`; or press `Ctrl-D`.

To interrupt a computation, type `Ctrl-C`.

Typing `Ctrl-C` twice within 30 seconds exits MAGMA immediately.

Expressions typed at the MAGMA prompt can span more than one line and will not be evaluated until you type a semi-colon (`;`) and press Enter (or Return).

Alternatively, you can use the MAGMA calculator at

<http://magma.maths.usyd.edu.au/calc>

Sets Sequences
Permutations
Functions Procedures

A simple problem

How can you use MAGMA to find an English word composed of the letters "a", "b", "c", "t", "r" ?

Well, you could look at all 120 permutations of the letters and hope to recognise which ones (if any) are English words.

First of all, enter the list into MAGMA as a sequence

```
> letters := ["a","b","c","t","r"]; // a sequence of strings of length 1
```

Use the symmetric group on $\{1, 2, 3, 4, 5\}$ to generate all permutations of the letters, then *concatenate* them to form 'words'.

```
> for p in Sym(5) do &*[ letters[i^p] : i in [1..5] ]; end for;
```

```
abctr
```

```
bctra
```

```
. . .
```

(Many commands in MAGMA have *synonyms*: e.g., `Sym` is a synonym of `SymmetricGroup`, and `SL` is a synonym of `SpecialLinearGroup`.)

Explanations

- `//` introduces a comment.
- We use a for-loop: `for ... do ... end for;` to iterate over the elements of the symmetric group.
- `letters[i]` refers to the i -th element of the sequence; indexing begins at 1 (not 0).
- `i^p` applies the permutation p to i .
- Strings are a monoid with binary operation `*`. If X is a sequence, `&*X` concatenates the elements of X . (You can also use `&cat X`.)
- `[1..5]` is the sequence `[1,2,3,4,5]`.
- `letters := ["a","b","c","t","r"];` is an *assignment statement*.
- MAGMA prints the results of *expressions* (such as `&*[letters[i^p]]`) that are not statements. Sometimes, for clarity, it is better to use the keyword `print`.

Checking the dictionary

Looking through 120 possible 'words' and then checking the dictionary doesn't seem like much fun.

Let's see how MAGMA can help. In this case it will depend on the operating system. Unfortunately, because of the `System` call, this won't run in the online calculator.

If MAGMA is running on MacOS or Linux you can do the following.

```
> letters := ["a","b","c","t","r"]; // a sequence of strings of length 1
> for p in Sym(5) do
>   word := &*[ letters[i^p] : i in [1..5] ];
>   cmd := "grep -w " cat word cat " /usr/share/dict/words";
>   System(cmd);
> end for;
bract
```

On Windows, find a word list somewhere (say `words.txt`), then use

```
> cmd := "findstr \"\\<\" cat word cat \"\\>\" words.txt";
```

More work

Suppose you modify the problem and ask for the *five* letter words composed of the letters "h","r","s","u","k","n".

For each subset of five letters, apply the previous solution.

```
> letters := ["h","r","s","u","k","n"];
> S := Sym(5);
> for n := 1 to 6 do
>   X := Remove(letters,n); // remove the n-th letter
>   for p in Sym(5) do
>     word := &*[ X[i~p] : i in [1..5] ];
>     cmd := "grep -w " cat word cat " /usr/share/dict/words";
>     System(cmd);
>   end for;
> end for;
```

hunks

Functions and procedures

Now suppose you want to solve the word puzzle for other combinations of letters.

Instead of typing ever more variations of the code into the REPL the thing to do is to create a function or procedure, store it in a file, then load the file whenever you need it.

In MAGMA a *function* takes *arguments* and returns one or more *values*. A *procedure* is similar except that it doesn't return any values.

The first step will be to write a procedure `findwds` (details on the next slide) in a file called `wordgame.m`. (You only need to print the result, so use a procedure, not a function.)

To load the file and use the procedure, type the commands

```
> load "wordgame.m";  
> findwds("abctr");  
bract
```

First version

```
> findwds := procedure(str)
>   letters := Eltseq(str); // change the string to a sequence
>   m := #letters;          // m is the number of letters
>   for p in Sym(m) do
>     word := &*[ letters[i~p] : i in [1..m] ];
>     cmd := "grep -w " cat word cat " /usr/share/dict/words";
>     System(cmd);
>   end for;
> end procedure;
```

MAGMA has a 'flat' namespace. The MAGMA kernel and all packages are loaded at startup. The functions and procedures (such as `Eltseq`) in these packages are called *intrinsic*s; they are always available.

```
> load "wordgame.m";
> findwds("torecv");
```

vector

covert

Second version

Modify the procedure to take both a *string* argument `str` and an *integer* argument `k` giving the length of the words we are looking for.

```
> findwds2 := procedure(str,k)
>   letters := Eltseq(str);
>   m := #letters;
>   if k gt m then k := m; end if;
>   R := Subsets({1..m},m-k);
>   S := Sym(k);
>   for A in R do
>     X := [ letters[i] : i in [1..m] | i notin A ];
>     for p in S do
>       word := &*[ X[i^p] : i in [1..k] ];
>       cmd := "grep -w " cat word cat " /usr/share/dict/words";
>       System(cmd);
>     end for;
>   end for;
> end procedure;

> findwds2("torxcv",4);
torc
```

Explanations

- if `ss` is a sequence, a set, a group, etc., `#ss` is its length.
- `if ... then ... else ... end if;`
- `if ... then ... elif ... else ... end if;`
- Sequence constructor: `[f(x) : x in O | condition on x]`
- Set constructor: `{ f(x) : x in O | condition on x }`
- `Set(L)` changes a sequence `L` to a set.
- `SetToSequence(S)` changes a set `S` to a sequence.

What happens if there are repeated letters?

```
> findwds2("tocrxc",4);
```

```
torc
```

```
torc
```

```
croc
```

```
croc
```

Exercise. Write a version of the code that removes duplicates.

Catalan numbers

Definition

The n th Catalan number is the number of balanced strings of n left and n right brackets, where 'balanced' means that each left bracket has a matching right bracket (to its right) and the string in between is balanced. The empty string is balanced.

We shall use MAGMA to construct sets of balanced strings. It turns out that

$$c_n = \frac{1}{n+1} \binom{2n}{n}.$$

Using MAGMA the 50th Catalan number is

```
> print Binomial(100,50) div 51;
```

Note that `div` is used for integer division and MAGMA can deal with very large numbers.

Real numbers

Using Stirling's approximation for $n!$ we have an asymptotic approximation

$$c_n \sim 2^{2n} / n \sqrt{\pi n}.$$

To use π in MAGMA we specify the precision.

```
> pi := Pi(RealField(10));
```

Here is a function which returns Stirling's approximation to the n th Catalan number.

```
> approxCat := func< n | 2^(2*n)/(n*Sqrt(pi*n)) >;  
> print approxCat(15);  
10427688.40
```

`func< x, y, z | expression >` defines a function which returns the value of the expression in the arguments x, y, z .

A recurrence relation

A balanced string of brackets is either empty or has the form $(S)T$, where S and T are themselves balanced. Therefore the Catalan numbers satisfy the recurrence relation (for $n > 0$)

$$c_{n+1} = \sum_{k=0}^n c_k c_{n-k} \quad \text{where } c_0 = 1.$$

Recursion in MAGMA

You can use `$$` to refer to a MAGMA function within its own body.

The following function uses recursion to return the sequence of the first n Catalan numbers.

```
> CatSeq := function(n);
>   if n eq 0 then   seq := [1];
>   elif n eq 1 then seq := [1,1];
>   else
>     seq := $$ (n-1);
>     Append(~seq, &+[seq[k+1]*seq[n-k] : k in [0..n-1]]);
>   end if;
>   return seq;
> end function;
```

Instead of `$$` you can place the directive

```
> forward CatSeq;
```

in your file, somewhere before the function definition. Then you can refer to `CatSeq` within its own definition.

More explanations

- Procedures can modify their arguments. Such an argument is prefixed with a tilde (\sim) both in the definition and when called.
- The command `Append(\sim seq,num)` is a call to the intrinsic procedure `Append` that modifies `seq` by including `num` in the set.
- Sequences can be indexed by other sequences.

```
X := letters[Setseq(A)];
```

- Errors

```
> CatSeq(A)
```

```
>> CatSeq(A);
```

```
^
```

```
User error: Identifier 'A' has not been declared or assigned
```

Exercises

Exercise. Write a function expression `CatNum` that returns the n th Catalan number.

The expression `y := (n > 1) select 11 else 0;` assigns `11` to `y` if `n` is greater than `1` otherwise it assigns `0` to `y`.

Exercise. Rewrite `CatSeq` as a function expression using `select`.

Random processes

$$S \rightarrow (S)S$$

$$S \rightarrow \varepsilon$$

is a grammar that describes balanced strings. We can use this to design a procedure that displays a random balanced string.

```
> produce := procedure()  
>   seq := ["S"];  
>   rhs := ["(", "S", ") ", "S"];  
>   X := { 1 };  
>   repeat  
>     i := Random(X);  
>     if Random(1) gt 0 then Insert(~seq, i, i, rhs);  
>       else Remove(~seq, i); end if;  
>     X := { i : i in [1..#seq] | seq[i] eq "S"};  
>   until IsEmpty(X);  
>   print #seq gt 0 select &*seq else "eps";  
> end procedure;
```

Recursion in sequences

A recurrence relation for the Legendre polynomials $P_n(z)$ is

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

To set this up in MAGMA we need the polynomial ring in the indeterminate z over the rational numbers. We don't need to keep a name for the ring itself so we use the underscore character.

```
> _<z> := PolynomialRing(Rationals());
```

The MAGMA function `Self(n)` refers to the n th entry of a sequence within its constructor. The following code creates the sequence of the first 8 Legendre polynomials.

```
> 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[7];  
231/16*z^6 - 315/16*z^4 + 105/16*z^2 - 5/16
```

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

Geometry

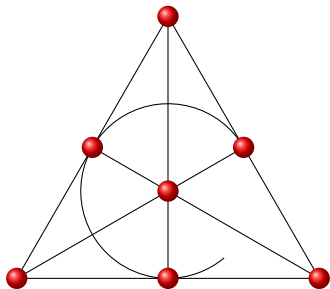
Graphs

Groups

The 7-point plane

A *projective plane* consists of a set of points and a set of lines such that every pair of distinct points lies on a unique line and every pair of distinct lines meet in a unique point.

The smallest projective plane is the *Fano* plane whose points are the 1-dimensional subspaces and whose lines are the 2-dimensional subspaces of a vector space of dimension 3 over the field of 2 elements.



Projective planes in MAGMA

If the plane is finite, there is an integer n (the *order* of the plane) such that every line has $n + 1$ points and every point lies on $n + 1$ lines. Thus there are $n^2 + n + 1$ points and $n^2 + n + 1$ lines.

The Fano plane is the unique projective plane of order 2.

```
> fano := FiniteProjectivePlane(2);  
> P := Points(fano);  
> L := Lines(fano);
```

The points and lines are represented by their (normalised) homogeneous coordinates.

```
> p := P[2];  
> l := L[3];  
> p; l, p in l;  
( 0 : 1 : 0 )  
< 0 : 0 : 1 >  
true
```

Defining a graph from the Fano plane

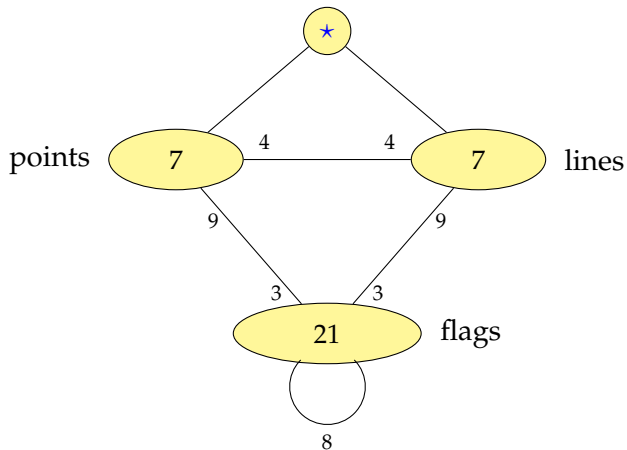
A *flag* in a projective plane is a point-line pair (p, ℓ) such that p lies on ℓ .

We shall construct a graph whose vertices are the points P , the lines L and the flags F of the Fano plane plus an additional vertex \star .

- Join \star to all of P and L .
- Join a point to the 4 lines not through it.
- Join a point to the 9 flags which have their line through it.
- Join a line to the 9 flags which have their point on it.
- Join flags (p_1, ℓ_1) and (p_2, ℓ_2) if $p_1 \neq p_2$, $\ell_1 \neq \ell_2$, $p_1 \in \ell_2$ and $p_2 \in \ell_1$.

The graph

36 vertices
degree 14
252 edges



Creating the graph in MAGMA : vertices

To build a graph in MAGMA we need a set of *vertices* and a set of *edges*.
In an undirected graph the edges are pairs of vertices.

For the graph from the Fano plane we represent the vertices as pairs of integers.

Represent

- \star by the pair $\langle 0, 0 \rangle$,
- the point $P[i]$ by $\langle -1, i \rangle$,
- the line $L[j]$ by $\langle -2, j \rangle$,
- the flag $(P[i], L[j])$ by $\langle i, j \rangle$.

```
> vertices := { <0,0> }  
>   join { <-1,i> : i in [1..7] }  
>   join { <-2,j> : j in [1..7] }  
>   join { <i,j> : i,j in [1..7] | P[i] in L[j] };
```

Creating the graph in MAGMA : edges

```
> F := [ <i,j> : i,j in [1..7] | P[i] in L[j] ];  
  
> edges := { {<0,0>,<-1,i>} : i in [1..7] }  
> join { {<0,0>,<-2,i>} : i in [1..7] }  
> join { {<-1,i>,<-2,j>} : i,j in [1..7] | P[i] notin L[j]}  
> join { {<-1,i>,<j,k>} : i,j,k in [1..7] | P[i] in L[k]  
>     and P[j] in L[k] }  
> join { {<-2,i>,<j,k>} : i,j,k in [1..7] | P[j] in L[k]  
>     and P[j] in L[i] }  
> join { {f,g} : f, g in F | f[1] ne g[1] and f[2] ne g[2]  
>     and (P[f[1]] in L[g[2]] or P[g[1]] in L[f[2]]) };
```

The graph constructor returns three values: the graph, the vertex set (type `GrphVertSet`) and the edge set (type `GrphEdgeSet`).

```
> Gr, V, E := Graph< vertices | edges >;
```

The automorphism group

```
> A := AutomorphismGroup(Gr);
> IsTransitive(A);
true
> CompositionFactors(A);
  G
  | Cyclic(2)
  *
  | 2A(2, 3)           = U(3, 3)
  1
> H := Stabiliser(A,1);
> CompositionFactors(H);
  G
  | Cyclic(2)
  *
  | A(1, 7)           = L(2, 7)
  1
> check, _ := IsIsomorphic(SU(3,3),DerivedGroup(A)); check;
true
```

Small Groups

Product-free sets

Let G be a group. A non-empty subset S of G is *product-free* if $ab \notin S$ for all $a, b \in S$.

Which finite groups have a maximal (by inclusion) product-free set of size 1, of size 2, of size 3, ...?

Checking if a set is product-free is easy.

```
> prodfree := func< S | forall{<a,b> : a,b in S | a*b notin S } >;
```

Finding *all* the groups with a product-free set of size 1, 2 or 3 is harder.

Looking for maximal product-free sets

We'll start with maximal product-free sets of size 1.

```
> checkmax1 := function(G)
>   for a in G do
>     if a eq One(G) then continue; end if;
>     found := true;
>     for b in G do
>       if b eq One(G) or b eq a then continue; end if;
>       if prodfree({a,b}) then found := false; continue; end if;
>     end for;
>     if found then return true, a; end if;
>   end for;
>   return false, _;
> end function;
```

Let's check a few cyclic groups.

```
> [checkmax1(CyclicGroup(n)) : n in [2 .. 10]];
[ true, true, true, false, false, false, false, false, false ]
```

Looking further

The results so far suggest that the only cyclic groups containing a product-free set of size 1 are C_2 , C_3 and C_4 . (Actually, it's quite easy to prove this directly.)

Are there any other groups with a product-free set of size 1?

```
> [checkmax1(DihedralGroup(n)) : n in [3 .. 10]];
[ false, false, false, false, false, false, false, false ]
```

None there. So where else can we look?

Databases

MAGMA has a large number of databases containing information that may be used in searches for interesting examples or which form an integral part of certain algorithms.

Examples of current databases include factorisations of integers of the form $pn \pm 1$, p a prime; modular equations; strongly regular graphs; maximal subgroups of simple groups; integral lattices; K3 surfaces; best known linear codes and many others.

We shall use MAGMA's Small Groups Database to get a supply of groups to check for small product-free sets.

Perhaps we can spot a pattern that will lead to a proof of their classification.

Using the Small Groups Database

The number of groups of order n in the database is returned by

```
> NumberOfSmallGroups(n).
```

To extract a copy of the j -th group of order n use

```
> G := SmallGroup(n,j).
```

Find the groups of order at most 50 that contain a product-free set of size 1.

```
> for n := 2 to 50 do
>   for j := 1 to NumberOfSmallGroups(n) do
>     G := SmallGroup(n,j);
>     found, witness := checkmax1(G);
>     if found then print n,j,witness; end if;
>   end for;
> end for;
```

(For efficiency, first open the database, then pass the reference as the first argument to the database functions.)

```
> SGD := SmallGroupDatabase();
```

The structure of the groups

The output from the previous command is

```
2 1 G.1
3 1 G.1
4 1 G.2
8 4 G.3
```

We know that the groups of orders 2, 3 and 4 are cyclic. What is the structure of the group of order 8?

```
> G := SmallGroup(8,4);
> IsAbelian(G);
false
> G;
GrpPC of order 8 = 2^3
PC-Relations:
$.1^2 = $.3,
$.2^2 = $.3,
$.2^$.1 = $.2 * $.3
```

Coset action

To convert G to a permutation group we can look at its regular representation. This is equivalent to its action on the cosets of the identity subgroup.

```
> f, H, K := CosetAction(G, sub<G | >);
```

```
> f;
```

```
Homomorphism of GrpPC : G into GrpPerm: H, Degree 8 induced by
```

```
G.1 |-> (1, 6, 2, 5)(3, 8, 4, 7)
```

```
G.2 |-> (1, 4, 2, 3)(5, 7, 6, 8)
```

```
G.3 |-> (1, 2)(3, 4)(5, 6)(7, 8)
```

```
> H;
```

```
Permutation group H acting on a set of cardinality 8
```

```
(1, 6, 2, 5)(3, 8, 4, 7)
```

```
(1, 4, 2, 3)(5, 7, 6, 8)
```

```
(1, 2)(3, 4)(5, 6)(7, 8)
```

```
> K; // the kernel of f
```

```
GrpPC : K of order 1
```

```
PC-Relations:
```

Identifying the group

```
> #{ x : x in H | Order(x) eq 2 };
```

```
1
```

So H is the quaternion group.

```
> Q := Group< a, b | a^2 = b^2, b^a = b^-1 >;
```

```
> IsIsomorphic(PCGroup(Q),H);
```

```
true Mapping from: GrpPC to GrpPerm: H
```

```
Composition of Mapping from: GrpPC to GrpPC and
```

```
Mapping from: GrpPC to GrpPerm: H
```

Why can't we use `IsIsomorphic(Q,H);` ?