Introduction to Magma and Applications

David R. Kohel

School of Mathematics and Statistics
The University of Sydney

presentation at the

African Institute for the Mathematical Sciences
2 February 2005

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,

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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

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 by setting 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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

6. New structures from old

The construction of objects in Magma is recursive, we can rational function fields over the integers, create an elliptic curve over the function field, and compute the function field of the curve.

```
> F1<u> := FunctionField(ZZ);
> F2<x> := FunctionField(F1);
> E := EllipticCurve([u+1,u,u,0,0]);
> E;
Elliptic Curve defined by y^2 + (u + 1)*x*y + u*y =
x^3 + u*x^2 over Rational function field of rank 1
over Integer Ring
Variables: u
> P := E![0,0,1];
> P;
(0 : 0 : 1)
```

6. New structures [a universal curve]

Now we can do arithmetic in this curve – even though no one specifically designed function fields to be used as base fields for elliptic curves.

```
> [ k*P : k in [1..4] ];
[ (0 : 0 : 1), (-u : u^2 : 1), (-u : 0 : 1),
(0 : -u : 1) ]
> 5*P;
(0 : 1 : 0)
> $1 eq E!0;
```

N.B. The above example shows that modular curve $X_1(5)$ has genus 0. The elliptic curve `E` above is a universal curve over $X_1(5)$.

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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

7. An exercise with elliptic curves

Exercise. Construct the generic point on the elliptic curve

$$y^2 + y = x^3 - x^2 - 10x - 20$$

over its own function field, and determine expressions for multiplication-by-n on the curve.

```
> E := EllipticCurve([0,-1,1,-10,-20]);
> E;
Elliptic Curve defined by y^2 + y =
x^3 - x^2 - 10*x - 20 over Rational Field
> Kx<x> := FunctionField(ZZ);
> Py<y> := PolynomialRing(Kx);
> fE := y^2 + y - (x^3 - x^2 - 10*x - 20);
> KE<y> := quo< Py | fE >;
> P := E(KE)![x,y,1];
> P;
(x : y : 1)
```

7. An exercise with elliptic curves [cont]

Note that the fact that we are allowed to create the point $P = (x, y)$ means that its coordinates satisfy the defining equation of the curve. This point P on E is a generic point of the curve.
The group law on E lets us compute the duplicate of this point:

```
> 2*P;
((x^4 + 20*x^2 + 158*x + 21)/(4*x^3 - 4*x^2 - 40*x - 79) :
    (2*x^6 - 4*x^5 - 100*x^4 - 790*x^3 - 210*x^2 - 1496*x -
    5821)/(16*x^6 - 32*x^5 - 304*x^4 - 312*x^3 + 2232*x^2 +
    6320*x + 6241)*y + (-7*x^6 + 14*x^5 + 102*x^4 - 239*x^3 -
    1221*x^2 - 3908*x - 6031)/(16*x^6 - 32*x^5 - 304*x^4 -
    312*x^3 + 2232*x^2 + 6320*x + 6241) : 1)
```

The denominator of the coordinates vanishes on the 2-torsion points of the curve; they are each a power of the second *division polynomial* on the elliptic curve E.

7. An exercise with elliptic curves [cont]

We can determine the denominator polynomial for higher multiple:

```
> x5 := (5*P)[1]; // the first coordinate of 5*P
> div5 := Denominator(Eltseq(x5)[1]);
25*x^24 - 200*x^23 - 5640*x^22 - 42890*x^21 +
  753166*x^20 + 8168640*x^19 + 28835955*x^18 -
  215272490*x^17 - 656490945*x^16 - 1427718412*x^15 -
  23616443060*x^14 - 55457442820*x^13 +
  250251082205*x^12 + 1401596289900*x^11 +
  3894316230546*x^10 + 17329402061420*x^9 +
  5672275314141 5*x^8 + 28542788370500*x^7 -
  350397547132965*x^6 - 1006316129964200*x^5 -
  7455234452899 25*x^4 + 13129344897767 50*x^3 +
  3561649459187025*x^2 + 3327725601836000*x +
  1319113344160000
```

7. An exercise with elliptic curves [cont]

From its factorization we find that the 5-torsion is very special:

```
> Factorization(div5);
[
    <x - 16, 2>,
    <x - 5, 2>,
    <5*x^2 + 5*x - 29, 2>,
    <x^4 + x^3 + 11*x^2 + 41*x + 101, 2>,
    <x^4 + 15*x^3 + 120*x^2 + 200*x + 155, 2>
]
```

In particular, the x coordinates 5 and 16 corresponds to 5-torsion point on the curve E.

```
> Q := E![5,5,1];
> {@ n*Q : n in [1..5] @};
{@ (5 : 5 : 1), (16 : -61 : 1), (16 : 60 : 1),
(5 : -6 : 1), (0 : 1 : 0) @}
```

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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

8. 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.

8. 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`.

8. 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
```

8. 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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

9. 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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

10. 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
```

10. 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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

11. 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)

11. 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/,

11. 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.

11. 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.

11. 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. An exercise with elliptic curves.
8. Built-in operators.
9. Language syntax.
10. Functions and procedures.
11. Packages and intrinsics.
12. An more advanced example.

12. A more advanced example

The Selmer curve is a classical example of a genus 1 curve having no rational points over $\mathbb{Q}$.

```
> P2<X,Y,Z> := ProjectiveSpace(QQ,2);
> C := Curve(P2,3*X^3+4*Y^3+5*Z^3);
> C;
Curve over Rational Field defined by
3*X^3 + 4*Y^3 + 5*Z^3
```

despite having points over every completion $\mathbb{Q}_p$ (and $\mathbb{R}$) for which we only need to check 2, 3, and 5:

```
> IsLocallySolvable(C,2);
true (1 + O(2^50) : O(2) : 1 + O(2))
> IsLocallySolvable(C,3);
true (O(3^3) : -2 + O(3^2) : 1 + O(3^50))
> IsLocallySolvable(C,5);
true (-2 + O(5) : 1 + O(5^50) : O(5))
```

Thus by an explicit computation we can verify the failure of the Hasse–Minkowski principle for this curve.

12. A more advanced example [cont]

We can actually construct the Jacobian E of this curve together with a map $C \to E$:

```
> E := EllipticCurve([0,900]);
> m := map< C->E |
> [ 20*X*Y*Z, 10*(4*Y^3-5*Z^3), -X^3 ] >;
```

Note that the lack of errors in creation of this map is a check that the given functions define a valid map of curves. We can moreover verify that $E(\mathbb{Q}) \cong \mathbb{Z}/3\mathbb{Z}$:

```
> G, phi := MordellWeilGroup(E);
> G;
Abelian Group isomorphic to Z/3
Defined on 1 generator
Relations:
    3*G.1 = 0
```

12. A more advanced example [cont]

The second argument, `phi` is the isomorphism $G \to E(\mathbb{Q})$. We can apply this map to the generator of G to find a generator for the rational points of $E(\mathbb{Q})$.

```
> P := phi(G.1);
> P;
(0 : 30 : 1)
> P, 2*P, 3*P;
(0 : 30 : 1) (0 : -30 : 1) (0 : 1 : 0)
```

12. A more advanced example [cont]

Having determined the three points of $E(\mathbb{Q})$, we can check that none of its points is the image of a rational point point of $C(\mathbb{Q})$. The syntax `P@@m` defines the pullback by `m` of a point `P` as a subscheme of the domain of `m`.

```
> P@@m;
Scheme over Rational Field defined by
X*Y*Z,
3*X^3 + 4*Y^3 - 5*Z^3,
3*X^3 + 4*Y^3 + 5*Z^3
> Degree($1);
3
> &join[ RationalPoints((n*P)@@m) : n in [1..3] ];
{@ @}
```

This completes the proof that $C/\mathbb{Q}$ itself has no rational point over $\mathbb{Q}$.

12. A more advanced example [cont]

The same curve can be defined over a finite field for cryptographic use.

```
> p :=
> 14615016373309029182036848327162830196593254347;
> // There exists a ‘‘twist’’ of the curve E/F_p which
> // has the prime number of points:
> N :=
> 14615016373309029182036827996659022319581997495 9;
> E := EllipticCurve([ FiniteField(p) | 0, 9000 ]);
> F := [ A : A in Twists(E) | #A eq N ][1];
> F;
Elliptic Curve defined by y^2 = x^3 + 18000 over
GF(14615016373309029182036848327162830196593254347)
> P := F![-20, 100, 1];
> Order(P);
14615016373309029182036827996659022319581997495 9
```

This gives a curve over a finite field whose group of points is a cyclic group of prime order, suitable for cryptographic applications.