16
Programming MAGMA

MAGMA is an excellent tool for computations of an algebraic nature, e.g., finite group theory, combinatorics, computations with basic number theoretic objects, and working with elliptic curves. However, my TI-89 hand calculator is better at computing integrals than MAGMA.

MAGMA has good support for developing large programs and combining code from many projects together. MAGMA's rigorous approach to computer algebra avoids much of the ambiguity that affects some other systems, and forces the user to produce more meaningful code that is easier to read and quicker. MAGMA also has highly optimized support for linear algebra over the rational numbers and $\mathbb{Z}/p$.

This chapter focuses on what MAGMA is and how to use it as a tool to accomplish more than a few quick computations in the shell. We do not dwell on specific MAGMA packages or functions.

16.1 Documentation

Thousands of pages have been written about MAGMA:

`http://magma.maths.usyd.edu.au/magma/htmlhelp/doc.htm`

Invest an hour and read the 12-page *First Steps in MAGMA*, then skim through the 884-page *Introduction*.

Instead of using the help system that is built into the MAGMA shell, I use the HTML reference manual. To look up a command, go to the index for the first letter of the command, then use your browser's find function to find the command, then click on the link. This will lead you to the help for the command, and you can easily navigate up in order to get information about how that command fits in with other commands.

You can also get documentation about the ways to call a command by typing its name, for example:

```
> PolynomialRing;
Intrinsic 'PolynomialRing'
Signatures:
(<RngInvar> R) -> RngMPol
The generic polynomial ring in which the elements of R lie
(<Rng> R) -> RngUPol
[
Global: BoolElt
]
Create the univariate polynomial ring over R
[... etc. for a page]
```

Notice that the behavior of `PolynomialRing` depends on the type of argument you give it.

16.2 Getting Comfortable

Once installed, if you run MAGMA you get a shell in which you can type commands. Without some customization, you will probably soon become impatient with the shell. You should do the following:

1. Create a directory, `magma` say, in which you will store MAGMA files.
2. Create a startup file, e.g., `startup.m`, which will be executed when you start MAGMA. (See Section 16.2.1.)
3. Create a `spec` file, which lists the filenames of code that you want to *attach* attach to MAGMA. (See Section 16.2.2.)
4. Learn to log your sessions to a file, and save and restore them. (See Section 16.2.3.)
5. If you want to use the MAGMA shell under another editor like the emacs shell window, type the command `SetLineEditor(false);` into MAGMA.

16.2.1 Startup File

MAGMA assumes nothing. Some new MAGMA users are frightened when they do the following:

```
[joesixpack@couch]# magma
Magma V2.9-11   [...]
> f := x^2 + 1;
>> f := x^2 + 1;
        ^
User error: Identifier 'x' has not been declared or assigned
```

Like in many strongly typed languages, it is necessary to define `x` first.

```
> R<x> := PolynomialRing(RationalField());
> R;
Univariate Polynomial Ring in x over Rational Field
> f := x^2 + 1;
```

Next, you might be put off by having to type huge words like

`PolynomialRing` and `RationalField`,

but this source of frustration can also be easily circumvented:

```
> poly := PolynomialRing;
> Q := RationalField();
> R<x> := poly(Q);
> R;
Univariate Polynomial Ring in x over Rational Field
```

After typing those first two lines, for the rest of the session you can type `poly` wherever you would type `PolynomialRing` and `Q` where you would have typed `RationalField()`. To keep all of these customization from session to session, create a startup file. For example, make a file `startup.m` that contains the following lines:

```
poly := PolynomialRing;
Q := RationalField();
Z := IntegerRing();
R := RealField();
R<x> := poly(Q);
charpoly := CharacteristicPolynomial;
```

Then set the environment variable `MAGMA_STARTUP_FILE` to `startup.m` (with proper path). Henceforth whenever you start MAGMA, `Q` will be the rationals, and `charpoly` will be the same as `CharacteristicPolynomial`. (Note: When you use the MAGMA shell, if you press tab, MAGMA will do auto-completion.)

16.2.2 Spec File

As we will see in Section 16.3, the MAGMA programs you write are stored in files that you attach to MAGMA.

It is tedious attaching a file to MAGMA each time you start MAGMA, so if you set the environment variable `MAGMA_USER_SPEC` to `$HOME/magma/spec` and list the filenames to attach in `spec`, they will automatically be attached when you start MAGMA.

16.2.3 Loging, Saving, and Restoring

It's frustrating to do something using MAGMA, only to lose the steps of the computation because they've scrolled off the screen. Use the command `SetLogFile("logfile")`, which takes one argument, the name of a file, and appends a log of the current magma session to that file. Type `UnsetLogFile()` to turn off loging.

If you're in the middle of a MAGMA session, and would like to leave and come back to it later, type `save "session"` then quit MAGMA. After you restart MAGMA, type `restore "session"`. (Warning: If you install a new version of MAGMA, the session files you used under the previous version of MAGMA might not load anymore.)

16.3 Programming

The MAGMA programming language resembles many standard procedural languages. Code is divided into files, and the code in files are divided into "functions", "procedures", and "intrinsics". Functions have arguments and return a single value, like in many other languages. A procedure is exactly the same as a function, but it doesn't return a value. Whereas functions and procedures have file scope, intrinsics are exported to the MAGMA shell, and are indistiguishable to the user from any of the other built in MAGMA commands. When you write an intrinsic you extend the MAGMA shell.

Let's extend MAGMA by adding a command called `MySqrt` that computes a square root of any square in $\mathbb{Z}/p$. (This is for fun, since the built in command `IsSquare` already does this.) First create a file called `mysqrt.m` that contains the following lines.

```
function alg3(a)
   assert Type(a) eq RngIntResElt;
   p := Modulus(Parent(a));
   assert p mod 4 eq 3;
   return a^((p+1) div 4);
end function;

function alg1(a)
   assert Type(a) eq RngIntResElt;
   p := Modulus(Parent(a));
   assert p mod 4 eq 1;
```

```
   F := Parent(a);
   R<x> := PolynomialRing(F);
   Q<x> := quo<R|x^2-a>;
   while true do
      z := Random(F);
      w := (1+z*x)^((p-1) div 2);
      if Coefficient(w,0) eq 0 then
         return 1/Coefficient(w,1);
      end if;
   end while;
end function;

intrinsic MySqrt(a::RngIntResElt) -> RngIntResElt
{The square root of a.  We assume that a has a square root
 and that a is an element of Z/p with p prime.}
   p := Modulus(Parent(a));
   if p eq 2 then
      return a;
   end if;
   if a eq 0 then
      return a;
   end if;
   require IsPrime(p) :
              "The modulus of argument 1 must be prime.";
   require KroneckerSymbol(Integers()!a,p) eq 1 :
              "Argument 1 must have a square root.";
   if p mod 4 eq 3 then
      return alg3(a);
   else
      return alg1(a);
   end if;
end intrinsic;
```

There are `assert` statements in the functions because MAGMA does no type checking for arguments to functions, so we have to fake it. Incidentally, we discover that elements of $\mathbb{Z}/p$ are of type `RngIntResElt` by creating an element in the shell and asking for its type:

```
> Type(ResidueClassRing(5)!1);
RngIntResElt
```

We don't pass the modulus p to `alg3` and `alg1`, because a is an element of $\mathbb{Z}/p$ so the function only needs to know a, since a knows $\mathbb{Z}/p$, in the sense that the `Parent` of a is $\mathbb{Z}/p$. To discover the `Modulus` command, I looked up `ResidueClassRing` in the MAGMA HTML documentation, then looked at nearby commands until I saw one called `Modulus`.

The `assert p mod 4 eq 3` line illustrates a healthy level of paranoia. The return line does the square root computation then returns it.

The function `alg1` computes the square root in the case $p \equiv 1 \pmod 4$. After the usual type checking assertion, we create the quotient ring

$$R = (\mathbb{Z}/p)[x]/(x^2 - a).$$

We then raise random elements of the form $1 + zx$ to the power $(p-1)/2$ until finding one of the form vx. The answer is then $1/v$.

Everything is tied together and exported to MAGMA in the intrinsic, which is the last part of the file. The declaration of the intrinsic gives the type of the arguments (multiple arguments are allowed), the return type (multiple return values are allowed), and a *mandatory comment* which must be given in braces. Note that non-intrinsic comments in MAGMA use the usual C++ syntax (/* and */ and //.) After the comment we use `if` statements to treat two special cases. The `require` statement makes certain assertions about the input; if they fail the corresponding error message is printed and execution stops.

To make use of our new function, add the line `mysqrt.m` to your `spec` file. When you start MAGMA the command `MySqrt` will automatically be available. Alternatively, instead of adding `mysqrt.m` to your `spec` file, you can type `Attach("mysqrt.m")` in MAGMA, but this only survives until you exit MAGMA.

If while running MAGMA you edit the file `mysqrt.m`, the changes automatically take affect. There is no need to restart MAGMA.

Here's an example session:

```
> Attach("mysqrt.m");
> R := ResidueClassRing(37);
>> MySqrt(R!13);
          ^
Runtime error in 'MySqrt': Argument 1 must have a square root.
> MySqrt(R!11);
14
> R!14^2;
11
> MySqrt(R!11);
23
> R!23^2;
11
> R := ResidueClassRing(31);
> MySqrt(R!7);
10
> R!10^2;
7
> MySqrt(R!11);
```

```
>> MySqrt(R!11);
          ^
Runtime error in 'MySqrt': Argument 1 must have a square root.
> MySqrt(R!19);
9
> R!9^2;
19
```

We can also try large primes to see if the algorithm is at all efficient.

```
> p := NextPrime(0495959487929484949494928292049494 8913);
> p mod 4;
1
> R := ResidueClassRing(p);
> time MySqrt(R!5);              // time times the command
45065146537549164856318874663544 0563
Time: 0.150
> $1^2;                          // $1 means the last output.
5
```

16.4 Weaknesses

I think the biggest weakness of MAGMA is that it is a strongly typed language with some object orient constructions, but it is almost impossible for the user to define new data types. The types that come with MAGMA are extensive, including rings, fields, vector spaces, modules, elliptic curves, etc., but they don't begin to encompass all of mathematics. Users cannot derive from built in types, though the `declare attributes` construction provides a very weak form of derivation. There are two unsatisfactory ways to add a new type to MAGMA: one is to become friends with the developers of MAGMA and ask them to add a new type to the kernel; the other is to find out about secret "Hack objects", which are blank types with names like "Xxx" that are compiled into MAGMA with no predefined functionality.

A mild frustration with MAGMA is that it is impossible to read in MAGMA code from a running MAGMA program. This makes saving and later reloading MAGMA variables very tedious. The best way to get around this is to use another language, such as python or perl, to set up computations, then run MAGMA via a system call.

The core of MAGMA is closed source, which is undesirable in a tool that one wants to use to prove rigorous mathematical results. This can be frustrating, but fortunately MAGMA is nonprofit and the developers of MAGMA are friendly and responsive, so it is possible to find out what MAGMA is really doing in any particular situation by asking them.

MAGMA is a huge piece of software; for example, there are nearly 250000 lines of package code included with MAGMA, and the C-kernel of MAGMA is much larger than that. Dozens of people from around the world have contributed code to Magma, at various levels, and there are unfortunately still many bugs. That said, the Magma developers are responsive and quickly fix the bugs and send out patches to users.