Magma

D. Dakota Blair, B. Lutes, J. Pattillo, J. Teslow

April 4, 2007

1 Introduction to CAS and Magma

A Computer Algebra System (CAS) is a piece of software capable of performing symbolic computations as well as the traditional numerical calculations that accompany mathematics. They are high-level computer languages that allow a mathematician to perform most common mathematical computations in a single command. While not as versatile as a general programming language, they are extremely efficient at executing their built-in commands and thus are vastly superior coding languages for mathematicians without expertise in creating optimal programs.

The most common uses of computer algebra systems are in performing tedious symbolic computations and in providing graphical visualization for assistance in problem solving. Thus in assessing a computer algebra system, efficiency and ability in performing these two tasks must be the primary consideration. Some of the general computer algebra systems most often employed to perform these tasks are Mathematica, Maple, Macsyma, Reduce, Derive, Matlab, and Magma. Though the pros and cons of most of these computer algebra systems are discussed extensively in “Computer Algebra Systems: A Practical Guide” by Michael J. Wester, Magma is not among these. Therefore to get a feel for where it is most useful, we examine its design.

The computer algebra system Magma was designed to perform computations in “algebra, number theory, geometry, and algebraic combinatorics” ([4] 346). Within these broad categories, magma is specifically designed to handle computational problems in “group theory, algebraic number theory, linear algebra and module theory, sparse matrices, lattices, commutative algebra, representation theory, invariant theory, lie theory, algebraic geometry, arithmetic geometry, finite incidence structures, cryptography, coding theory, and optimization."

When working with the language, it is clear that Magma is slightly less versatile than other computer algebra systems. For instance, Magma does not perform graphical visualization operations so a traditional purpose for using a computer algebra system is conspicuously missing. However, Magma makes up for its lack of versatility by doing what it does well. For instance, since 2004, Magma has become the fastest program in the world at performing Polynomial GCD [5]. Likewise, Magma is extremely fast at computing Grobner bases, as is evidenced by its win in the 80 variable HFE Challenge in August 2004 by solving a system of 80 quadratic equations in 80 variables over GF(2) in 22.1 hours [5].

So how does Magma compare with PARI? PARI is designed soley for number theoretic calculations whereas Magma is designed to handle problems in a number of other disciplines. PARI is extemely fast on most number theoretic calculations, but is notably slow, compared to the more sophisticated general computer algebra systems as mentioned above at multivariate polynomials and formal integration[6].

There are still other factors besides performance to consider when comparing PARI and Magma. For instance, PARI is open source and simple to download and implement into a computing environment whereas Magma requires a licence. On the other hand, Magma has extensive documentation located at

http://www.msri.org/about/computing/docs/magma/html/MAGMA.htm

Pari has extensive documentation as well but nowhere is there an alphabetical index of the commands of the language and how to use them such as there is with Magma. Even so the source code is available so it is easier to see exactly how certain operations are carried out.

While the best algebra system to use in a given situation depends on more than simply what you want to calculate it is not always easy to determine. Hopefully this introduction illustrates where Magma would be a good choice. For the rest of this paper, the area of interest is primality testing. To illustrate the basics of Magma, we will implement the AKS primality test, in hopes of showing a small, but representative sample of the power of Magma.

2 PRIMES up to and including AKS

From Euclid's transcribed proof to the Riemann-Zeta hypothesis, it is possible that there is no set in all mathematics which has been studied more than that of PRIMES, the set of all prime numbers. Primality testing is the only way in which one may populate this set, and yet since the Sieve of Eratosthenes there was no known computationally efficient way of determining whether a given number was prime or not. Wilson's theorem (p is prime if and only if $(p-1)! \equiv -1 \mod p$) would be an ideal test if the computer could deal with arbitrarily large integers, but $n!$ grows too fast for any traditional computer model to deal with for large n. Further discussion on the primality testing problem itself see [1]. Another computational problem which is dependent upon PRIMES is that of prime factorization. It is still unknown how difficult this problem is, but if **P≠NP** it could be out of reach of computing as we know it. However, this problem certainly depends on primality testing, since the first step of prime factorization should eliminate the possibility that the number to be factored is itself a prime power.

In his *A Mathematician's Apology* Hardy says, "No one has yet discovered any warlike purpose to be served by the theory of numbers or relativity, and it seems unlikely that anyone will do so for many years." While this statement is no longer entirely accurate, the study of PRIMES itself is still considered by many to be the purest form of mathematics. The modern applications of large primes to cryptography has increased the interest in (comparatively) large primes. The largest known primes are the Mersenne primes and the largest of these have been found through a distributed computing project known as *The Great Internet Mersenne Prime Search*[7]. Other than the pure interest in large primes, there is also a substantial popular interest. The Electronic Frontier Foundation is offering up to $250,000 for discovering large primes[8]. The RSA is also offering prizes for factoring large coprimes[9]. These of course are new reasons for fast primality testing and factorization [one of these is now possible]

3 First step to implementation

In this section we will discuss the AKS algorithm and some of the computational hurdles that were encountered during its implementation.

First some notation. $\mathbb{Z}/n\mathbb{Z}$ denotes the ring of integers modulo n. We use log for base 2 logarithms. Given $r \in \mathbb{N}$, $a \in \mathbb{Z}$ with $(a, r) = 1$, the *order of a modulo r* is the smallest number k such that $a^k \equiv 1 \pmod{r}$. It is denoted $o_r(a)$. *Euler's totient function* is denoted by $\phi(r)$. We use the symbol $O^{\sim}(t(n))$ for $O(t(n) \cdot \text{poly}(\log t(n)))$ where $t(n)$ is any function of n and poly returns a polynomial in terms of its argument. For example, $O^{\sim}(\log^k n) = O(\log^k n \cdot \text{poly}(\log\log n)) = O(\log^{k+\epsilon} n)$ for any $\epsilon > 0$.

Here is the algorithm:

1. If $n = a^b$ for $a \in \mathbb{N}$ and $b > 1$, output COMPOSITE.
2. Find the smallest r such that $o_r(n) > \log^2 n$.
3. If $1 < (a, n) < n$ for some $a \leq r$, output COMPOSITE.
4. If $n \leq r$, output PRIME.
5. For $a = 1$ to $\lfloor \sqrt{\phi(r)} \log n \rfloor$ do

   if $(X + a)^n \neq X^n + a$ in $(\mathbb{Z}/n\mathbb{Z})[x]/(x^r - 1)$, output COMPOSITE.
6. Output PRIME.

This is a polynomial-time algorithm (more specifically, the asymptotic time complexity is $O^{\sim}(\log^{15/2} n)$ [Theorem 5.3, AKS]) which returns PRIME if and only if n is prime [Theorem 4.1, AKS]. All of the computing time of this algorithm goes into Steps 1, 2 and 5. Let us briefly examine the contents of these steps.

For Step 1, our main source is a paper of Bernstein[3]. Here, all references are to this paper. The interested reader may refer to this for a more thorough exposition. As one can see by referring to our code, detecting perfect powers using Bernstein's methods involve quite a bit of work. The power detection problem can be solved using three subroutines.

1. Given n, x, k, test whether $n = x^k$.
2. Given n, k, test whether n is a kth power.
3. By running through the primes $p \leq \log n$, check for each p whether n is a pth power.

Bernstein calls these Algorithms C, K and X, respectively. Their output is described by the following:

1. (Lemma 9.2) Algorithm C prints the sign of $n - x^k$.
2. (Lemma 10.3) If n is a kth power, Algorithm K prints the kth root of n. Otherwise, the output is 0.
3. (Lemma 11.2) If n is a perfect power, Algorithm X prints a prime number p and a positive integer x such that $x^p = n$. Otherwise, the output is $(n, 1)$.

As the reader may have noticed, this perfect power algorithm is given separately and not included in our main algorithm, for which we employed the services of Magma. We thought that one of the best ways to illustrate the utility of Magma was by replicating Bernstein's multi-step process with the single Magma command `IsPower`. It is hoped that the reader will appreciate this simplification.

In our primality testing implementation, Step 2 is denoted by the procedure `GetSmallestR`. This is the easiest of the steps to implement because Magma can do computations in the ring $\mathbb{Z}/r\mathbb{Z}$ in a single command. It should be noted that the value of r found can be quite large. In particular, we have that $r \leq \max\{3, \lceil \log^5 n \rceil\}$ [Lemma 4.3, [1]].

Finally, we arrive at the linchpin of the algorithm, namely Step 5. For obvious reasons, we refer to this as `FreshmansDreamTest`. Here we encounter a significant slowdown in the pace of the algorithm due to the amount of modular polynomial exponentiation involved. Indeed, if n is prime then this last step must be implemented $\lfloor \phi(r) \log n \rfloor$ times, and for each implementation, the expression $(X + a)^n$ must be expanded in order to compare it to $X^n + a$. Obviously, for large n this causes a serious bottleneck in our computation. Bernstein gives a method to overcome this difficulty[2], but ultimately we decided not to include this in our algorithm. Instead, in keeping with our theme of showcasing the ability of Magma, we decided to use the command `Modexp` as opposed to Bernstein's approach. Whether our approach is time-saving or not is something we are unable investigate at this time. We note that this comment applies to our entire implementation and not just Step 5.

4 Power Detection Implementation

```
function lg(A)
// This is a base two logarithm function
    return Log(2,A);
end function;

function NumDig(A)
// Given a number A. This will calcluate the number of binary digits of A
    if A eq 0 then
        return 1;
    end if;

    Dig:=Floor(Log(2,Abs(A)));
    if(A-2^Dig lt 0) then
        while (A-2^Dig lt 0) do
            Dig:=Dig-1;
            if (A-2^Dig ge 0) then
                return Dig+1;
            end if;
        end while;
    else
        return Dig+1;
    end if;

end function;

function Trunc(A,precision)
// Truncate A to precision Binary digits of accuracy or
// NumDig(A) whichever is less.
    Digits:=NumDig(A);
    k:=Digits-precision;

    if (k le 0) then
        return A;
    end if;

    return Floor(A*2^(-k))*2^(k);
end function;

function Pow(b,x,k)
//find x^k to b digits of precision.
    //Step 1
    if (k eq 1) then
        return Trunc(x,b);
    end if;

    //Step 2

    if IsEven(k) then
        k2:=Floor(k/2);
        sq:=Pow(b,x,k2);
        return Trunc(sq^2,b);
    end if;

    //Step 3
    qm1:=Pow(b,x,k-1);
    return Trunc(qm1 * Trunc(x,b),b);
end function;

procedure AlgC(n,x,k,~value)
// Find the sign of n-x^k.
    f := Floor(lg(2*n));
    b := 1;

    while 1 eq 1 do
        result:=Pow(b+Ceiling(lg(8*k)),x,k);

        if (n lt result) then
            value := -1;
            break;
        elif (n ge result*(1+2^(-b))) then
            value := 1;
            break;
        elif (b ge f) then
            value := 0;
            break;
        end if;
        b := Minimum(f,2*b);
    end while;
end procedure;

procedure FindG(y,~g)
```

```
// Find a constant needed for AlgB
    g:=Ceiling(Log(y)/Log(2));
end procedure;



procedure AlgB(b,y,k,~z)
// Find z st z^k*y-1=0 if b is suff. small.
    B:=Ceiling(Log(66*(2*k+1))/Log(2));
    g:=0;
    FindG(y,~g);
    a:=Floor(-g/k);

    //Step 1
    z:=2^a+2^(a-1);
    j:=1;

    while (1 eq 1) do
    //Step 2
        //Now z=NRoot(j,y,k,~z);
        if (j eq b) then
            break;
        else
    //Step 3
            zk:=Pow(B,z,k);
            ytrunc:=ChangePrecision(1.0*y,B);
            r:=ChangePrecision(zk*ytrunc,B);
    //Step 4
            if (r le 0.9697265625) then
                z:=z+2^(a-j-1);
            end if;
    //Step 5
            if (r gt 1) then
                z:=z-2^(a-j-1);
            end if;
        end if;
        j:=j+1;
    end while;
end procedure;

procedure AlgN(b,y,k,~z)
// Find z st z^k*y-1=0 if b is larger.
    //Initialize
    CL:=Ceiling(Log(2*k)/Log(2));
    b_prime:=CL+Ceiling((b-CL)/2);
    B:=2*b_prime+4-Ceiling(Log(k)/Log(2));
    //note b_prime<b

    //Step 1
    z:=0;
    if (b_prime le Log(8*k)/Log(2)) then
        AlgB(b_prime,y,k,~z);
    else
        AlgN(b_prime,y,k,~z);
    end if;

    //Step 2
    r2:=ChangePrecision(1.0*z,B)*(k+1);

    //Step 3
    r:=Pow(B,z,k+1);
    r3:=ChangePrecision(1.0*r*ChangePrecision(1.0*y,B),B);

    //Step 4
    z:=ChangePrecision(1.0*(r2-r3)/k,B);
end procedure;

procedure NRoot(b,y,k,~z)
// Find z st z^k*y-1=0 to b units of precision.

    z:=0;
    if (b le Ceiling(Log(8*k)/Log(2))) then
        AlgB(b,y,k,~z);
    else
        AlgN(b,y,k,~z);
    end if;
end procedure;



procedure FindY(n,b,~y)
// Find y suff. precise.
    f := Floor(Log(2*n)/Log(2));

    y:=0;
    NRoot(3+Ceiling(f/2),n,1,~y);

end procedure;

procedure AlgK(n,k,y,~x)
// Find x st x^k=n if possible
    f := Floor(Log(2*n)/Log(2));
    b := 3+Ceiling(f/k);

    y:=0;
    FindY(n,b,~y);

    //Step 1
    r:=0;
    NRoot(b,y,k,~r);
    //Step 2
    x:=Floor(0.625+r);
    //Step 3
    if ((x eq 0) or (Abs(r-x) ge 0.25)) then
        x:=0;
    else
        r2:=0;
        AlgC(n,x,k,~r2);
        if (r2 ne 0) then
            x:=0;
        end if;
    end if;
end procedure;

procedure AlgX (n,~x,~k)
//Find x,k s.t n=x^k.
    f := Floor(Log(2,2*n));

    //Step 1
    y:=0;
    k:=0;
    NRoot(3+Ceiling(f/2),n,1,~y);
    //Step 2
    for i:=2 to f-1 do
//        if IsPrime(i) then
            x:=0; AlgK(n,i,y,~x);
            if (x gt 0) then
                k:=i;
                break;
            end if;
//        end if;
    end for;

    //Step 3
    if (k eq 0) then
        x:=n;
        k:=1;
    end if;
end procedure;
```

5 Implementation of AKS in Magma

```
procedure GetSmallestR(n,~r)
    R := RealField();
    Zr := Integers(r);
    nr := Zr ! n;
    o := R!Order(nr);
    logvalue := Log(n)/Log(2);
    while (o le (logvalue^2)) do
        r := r+1;
        while (GCD(n,r) ne 1) do
```

```
            r := r+1;
        end while;
        Zr := Integers(r);
        nr := Zr ! n;
        o := R!Order(nr);
    end while;
end procedure;

procedure FreshmansDreamTest(r,n,~primeifzero)
    logvalue := Log(n)/Log(2);
    testvalue := Floor(((EulerPhi(r))^(1/2))*logvalue);
    Zn := Integers(n);
    P<x> := PolynomialRing(Zn);
    PP := PolynomialRing(Zn);
    testpoly := P ! (x^r-1);
    for a in [1..testvalue] do
        f:= P ! (x+a);
        f1 := Modexp(f,n,testpoly);
        g1 := Modexp(x,n,testpoly)+a;
        h := f1-g1;
        if (h ne (P ! 0)) then
            primeifzero := 1;
        end if;
    end for;
    if (primeifzero ne 1) then
        primeifzero := 0;
    end if;
end procedure;

procedure AKSPrimalityTester(n)
```

```
primeifzero := -1;
if (IsPower(n)) then
    primeifzero := 1;
else
    r := 2;
    while (GCD(n,r) ne 1) do
        r := r+1;
    end while;
    GetSmallestR(n,~r);
    for a in [1..r] do
        g := GCD(a,n);
        if ((g gt 1) and (g lt n)) then
            primeifzero := 1;
        end if;
    end for;
    if (primeifzero ne 1) then
        if (n le r) then
            primeifzero := 0;
        else
            FreshmansDreamTest(r,n,~primeifzero);
        end if;
    end if;
end if;
if (primeifzero eq 0) then
    print "This number is prime.";
else
    print "This number is composite.";
end if;
end procedure;
```

References

[1] Agrawal, Manindra, Neeraj Kayal, and Nitin Saxena. "PRIMES is in P." Annals of Mathematics. 160 (2004) 781-793. 01 April 2007. <http://www.math.princeton.edu/ annals/issues/2004/Sept2004/Agrawal.pdf>.

[2] Daniel J. Bernstein. "Primality Testing After Agrawal-Kayal-Saxena." *Draft*. 160 (2004) 781-793. 01 April 2007. <http://cr.yp.to/papers/aks.pdf>.

[3] Daniel J. Bernstein. "Detecting Perfect Powers in Essentially Linear Time." Preprint, to appear in *Mathematics of Computation*. <http://cr.yp.to/papers/aks.pdf>.

[4] Wester, Michael J. Computer Algebra Systems: A Practical Guide. New York: John Wiley & Sons, 1999.

[5] "Magma computer algebra system." 04 April 2007. <http://en.wikipedia.org/wiki/Magma_computer_algebra_system/>.

[6] "PARI/GP." 04 April 2007. <http://en.wikipedia.org/wiki/PARI/GP>.

[7] *The Great Internet Mersenne Prime Search* 04 April 2007 <http://www.mersenne.org/>

[8] The Electronic Frontier Foundation Cooperative Computing Awards 04 April 2007 <http://www.eff.org/awards/coop.php>

[9] The RSA Challenge Numbers 04 April 2007 <http://www.rsa.com/rsalabs/node.asp?id=2093>