Gauss: a Parameterized Domain of Computation System with Support for Signature Functions

Michael B. Monagan

Institut für Wissenschaftliches Rechnen
ETH Zentrum, CH 8092 Zürich, Switzerland
monagan@inf.ethz.ch

Abstract. The fastest known algorithms in classical algebra make use of signature functions. That is, reducing computation with formulae to computing with the integers modulo p, by substituting random numbers for variables, and mapping constants modulo p. This idea is exploited in specific algorithms in computer algebra systems, e.g. algorithms for polynomial greatest common divisors. It is also used as a heuristic to speed up other calculations. But none exploit it in a systematic manner. The goal of this work was twofold. First, to design an AXIOM like system in which these signature functions can be constructed automatically, hence better exploited, and secondly, to exploit them in new ways. In this paper we report on the design of such a system, Gauss.

1 Introduction

In 1980 Schwarz [11] proposed the following *probabilitistic* method for testing if a matrix of polynomials in $x_1, x_2, ..., x_n$ over the integers is singular or not. This method formalized an idea that was already being used in an ad-hoc way for speeding up various calculations in computer algebra systems, and other places at the time.

Procedure TESTZERO

Input: an n by n matrix A over $\mathbf{Z}[x_1, ..., x_n]$, a failure tolerance ϵ, a degree bound d on $\det(A)$

Output: false implies $det(A) \neq 0$, true implies $det(A) = 0$ with probability $\geq 1 - \epsilon$

```
error ← 1
while error > $\epsilon$ do
    choose prime $p > d$
    $A' \leftarrow A$ modulo $p$
    for $i = 1..n$ choose $\alpha_i$ at random from $\mathbf{Z}_p$
    $A' \leftarrow A$ evaluated at $x_i = \alpha_i$ modulo $p$
    $D \leftarrow det(A')$ modulo $p$
    if $D = 0$ then error ← error $\times$ $d/p$ else output false
output true
```

If this procedure outputs "false" then it has proven that the matrix A is non-singular. If the procedure outputs "true" then the procedure may make an error. The idea behind this approach is it will make errors *controllably* low probability ϵ where typically, one would arrange to have $\epsilon < 10^{-50}$. Schwarz's contribution is firstly, a theorem that says that the error bound is met provided the prime p is larger than the degree the determinant, a polynomial in $\mathbb{Z}[x_1, ..., x_n]$, which can be bounded easily in advance. Secondly, if the primes p are chosen to be at least $> 2d$ so that the probability of error is decreased by at least a factor of 2 at each step, then the complexity of the method will be satisfactory from a theoretical viewpoint. In practice, p will be much larger than d, so that only a few iterations of the loop are required before a small probability of error is achieved. Another well known application of such a *probabilistic* algorithm is primality testing. We refer the reader to Rabin [7] and Solovay & Strassen [8] for two different probabilistic algorithms for primality testing.

The idea of using modular mappings in this way is not used much in computer algebra systems. That is a pity because for many specific problems, such as testing whether a linear system has a solution or not, this approach is often computationally the only hope for an answer. In [3], and [4], Gonnet extended the class of expressions for which signature functions were available from rational expressions in $x_1, x_2, ..., x_n$ over $\mathbb{Q}$ to include, roughly speaking, firstly, unnested exponentials, logarithms, and trigonometric functions, and secondly, simple roots. This work led to the routine `testeq` in Maple which tests to see if an expression is zero or not. However, the ideas are not used by the rest of Maple. There is no facility for testing if a matrix is singular or not. In [9] we have looked at signatures for algebraic numbers and algebraic functions. One of the methods for algebraic integers, is based on the ideas in [10]. We will sketch another method later in this paper.

In our experiments, we have attempted to build such signature functions in a more systematic manner, so they can be exploited more generally. To do this, we have designed an AXIOM like system [1] which runs on top of Maple. We have called our system *Gauss*. This paper is organized as follows. In Section 2 we present the design of Gauss. In Section 3 we show how signature functions are built into Gauss. In a subsequent paper, we will report in more detail on the actual signature functions that we use in [9]. In Section 4 we give some novel applications of signature functions in Gauss. We end with some conclusions and comments about Gauss.

2 Gauss

We have designed a system which supports parameterized types and parameterized abstract types. In AXIOM terminology, these are called *domains* and *categories* respectively. The principle motivation is to allow one to implement an algorithm in a *generic* setting. Let us motivate the reason for doing this and illustrate how this is done in Gauss with an example. Consider computing the determinant of a square Matrix over a field using the algorithm Gaussian elimination. In Gauss the code would look like this:

```
GaussianEliminationDeterminant := proc(A,n,F) local d,i,j,k,m;
    d := F[1];
    for k to n do
        for i from k to n while F['=']( A[i,k], 0 ) do od;
        if i>n then RETURN(0) fi;
        if i<>k then
            ... # interchange row i with row k
            d := F['-'](d)
        fi;
        d := F['*'](d,A[k,k]);
        for i from k+1 to n do
            m := F['/'](A[i,k],A[k,k]);
            for j from k+1 to n do
                A[i,j] := F['-'](A[i,j],F['*'](m,A[k,j]))
            od
        od
    od;
    RETURN( d )
end;
```

The key difference between this code and other coding styles is the parameter F, the field. It is a collection of all the operations for computing over the field called a *domain*. In the Maple [6] code above, we see that we are using a Maple table, a hash table, for representing a domain. Thus a domain in Gauss consists essentially of a table of Maple procedures.

2.1 Using Gauss

The above example gives the reader an idea of how one programs in Gauss. Before we give more details, we show first how one uses Gauss interactively, and make some initial comments about the functionality which all domains in Gauss provide.
The Gauss package is available in Maple V Release 2 after executing the command `with(Gauss)`.

```
> with(Gauss);
---------------------- Gauss version 1.0 ----------------------
Initially defined domains are Z and Q the integers and rationals

                                              [init]
```

Initially, the two domains `Z`, the integers $\mathbb{Z}$, and `Q`, the rationals $\mathbb{Q}$, have been defined. Let's do some operations with them:

```
> Z[Gcd](8,12);
                                                4
```

Principle 1: All named operations in Gauss begin with an upper case letter and must be "package-called", i.e. the domain subscript must be explicitly used.

```
> Q['+'](1/2,1/3,1/4);
                                        13
                                       ----
                                        12
> Z[Gcd];

                                        igcd
```

Principle 2: Gauss uses the Maple representation for integers and rationals, and the builtin Maple functions where appropriate, e.g. the `igcd` function. Since Gauss is written in Maple, i.e. Gauss code is Maple code which is interpreted, this starting point is rather crucial for efficiency.

Next we want to perform some operations in $\mathbb{Q}[x]$. First we have to create the domain of computation for objects in $\mathbb{Q}[x]$. This is done by calling a domain constructor, in this case the constructor `DenseUnivariatePolynomial` (or `DUP` for short). A domain constructor in Gauss is a Maple procedure (with zero or more parameters), which returns a Gauss domain. The domain constructor `DUP` takes two parameters, the coefficient ring and the name of the variable.

```
> D := DUP(Z,x):
> a := D[Random]();
                              a := [-37, -35, -55]
```

Principle 3: All domains in Gauss have a `Random` function which creates a "random" example from the domain.

```
> d := D[Diff](a);
                                 d := [-35, -110]

> D[Output](d);
                                   - 110 x - 35
```

Principle 4: All domains in Gauss support an `Output` function which converts from the domains internal data representation to the Maple form.

```
> m := D[Input](y^4-10*x^2-y);

                                    m := FAIL

> m := D[Input](x^4-10*x^2-1);

                             m := [-1, 0, -10, 0, 1]
```

Principle 5: All domains in Gauss support an `Input` function which tries to convert a Maple expression into the Gauss representation. We do require that `D[Input](D[Output](x))` always succeeds as this is also used as a general mechanism for converting from one data type to another.

```
> D[Type](m);

                                        true
```

`Principle 6`: Since Gauss is implemented on top of Maple, there is no means for static type checking. Type checking is therefore done at run-time in a Maple style manner. Note, we are not advertising this particular feature as a "good" feature of Gauss.

In the remaining parts of this section on Gauss, we will assume that the reader is somewhat familiar with the AXIOM approach. We will present Gauss from AXIOM's point of view, due to lack of space. As we go, we will point out differences.

First, we wish to give some reasons for why we have designed another system instead of using AXIOM. AXIOM suffers from being big and inflexible. If you want to make major changes to fundamental domains, then you will have a lot of work to do. Especially because you would have to recompile dependencies, and the compiler is very slow. If you want to experiment with this, it will be too painful to bear. What we really needed was a small flexible system. That is essentially what Gauss is.

2.2 Domains and Categories in Gauss

Domains (parameterized types) are created by executing Maple functions. The parameters to a domain can be values and other domains. We have seen how domains are used in the determinant example.

Categories (parameterized abstract types) are just domains with missing functions. They are also parameterized by values and domains. Their purpose is to provide code which does not depend on the data representation, and hence can be shared amongst more than one domain.

These ideas, how it is done in Gauss, and the differences with AXIOM are best shown by looking at carefully chosen examples. In Gauss, here is the definition for the category Set, the category to which all other categories belong. The call `Set()` creates a domain which belongs to the category Set. It defines what operations a domain which is a Set must have, and implements one of them. The call `Set(S)` extends a domain by adding these operations to the domain set.

```
Set := proc() local S;
    if nargs = 0 then S := newCategory() else S := args[1] fi;
    if hasCategory(S,Set) then RETURN(op(S)) fi;
    addCategory(S,Set);
```

Specifies that S, the domain being constructed, belongs to the category Set.

```
    defOperations( {'=','<>'}, [S,S] &-> Boolean, S );
```

Adds the definitions for the two operations equality and inequality which have the signature indicated to the domain being constructed S.

```
    defOperation( Random, [] &-> S, S );
    defOperation( Input, Expression &-> Union(S,FAIL), S );
    defOperation( Output, S &-> Expression, S );
    defOperation( Type, Expression &-> Boolean, S );
    S['<>'] := subs(D = S,proc(x,y) not D['='](x,y) end);
```

Adds a default definition for the operation inequality in terms of equality.

```
      S[Output] := x -> x; # use Maple by default
      op(S)
  end:
```

The Euclidean Domain in Gauss is

```
  EuclideanDomain := proc() local E;
      if nargs = 0 then E := newCategory() else E := args[1] fi;
      E := UniqueFactorizationDomain(E);
```

Inherits (adds to E) all the operations defined and implemented from the category UniqueFactorizationDomain.

```
      addCategory(E,EuclideanDomain);
      defOperation( EuclideanNorm, E &-> Integer, E );
      defOperation( SmallerEuclideanNorm, [E,E] &-> Boolean, E );
      defOperations( {Rem,Quo}, {[E,E] &-> E,[E,E,Name] &-> E}, E );
      defOperation( Gcdex, {[E,E,Name,Name] &-> E,[E,E,Name] &-> E}, E );
      defOperation( Powmod, [E,Integer,E] &-> E, E );
      E[Quo] := subs('D' = E, proc(x,y,r) local t,q;
          t := D[Rem](x,y,q); if nargs = 3 then r := t fi; q
          end);
      E[Div] := subs('D' = E, proc(x,y) local q;
          if D[Rem](x,y,q) <> D[0] then FAIL else q fi
          end);
      E[SmallerEuclideanNorm] := subs('D' = E, proc(x,y)
          evalb( D[EuclideanNorm](x) < D[EuclideanNorm](y) )
          end);
      E[Powmod] := subs('D' = E, proc() PowerRemainder(D,args) end);
      E[Gcd] := subs('D' = E, proc() EuclideanAlgorithm(D,args) end);
```

The default algorithm for computing GCD's is the Euclidean Algorithm. It is implemented in the same manner as the GaussianElimination algorithm.

```
      E[Gcdex] := subs('D' = E, proc() PrincipalIdeal(D,args) end);
      op(E)
  end:
```

Notice that of the new operations defined in the EuclideanDomain category, only the Euclidean norm and remainder operations are not defined. The other operations are defined either in-line, or as calls to out-of-line procedures. We include here the code for the EuclideanAlgorithm, because this code illustrates the way one implements algorithms in Gauss, and it is probably the best generic implementation possible of the Euclidean algorithm. Note that the algorithm is n-ary, i.e. it computes the GCD of zero or more arguments.

```
  EuclideanAlgorithm := proc(E) local a,b,r,s;

      s := {args[2..nargs]} minus {E[0]};
```

A property of Gauss inherited from Maple is that for Gauss domains which have canonical forms, equal values are represented uniquely, and hence here, duplicates can be removed very efficiently.

```
      if s = {} then RETURN(E[0]) fi;
      sort( [op(s)], E[SmallerEuclideanNorm] );
```

The idea here is to start with the smallest values, because the size of GCD's always get smaller in the Euclidean norm sense, and usually, they this carries over to the computational cost too.

```
        a := E[Normal](s[1]);
        for b in subsop(1=NULL,s) while a <> E[1] do
            b := E[Normal](b);
            while b <> E[0] do
            r := E[Normal](E[Rem](a,b));
```

This is the monic Euclidean algorithm. That is, the Normal function here is making the remainder unit normal. This is a generic attempt to control intermediate expression swell.

```
            a := b;
            b := r
            od;
        od;
        a
    end:
```

We continue with an outline of the category for univariate polynomials.

```
      UnivariatePolynomial := proc() local P,R,env,U,'?';

        R := args[1];
        if not hasCategory(R,Ring) then
            ERROR('1st argument must be a ring') fi;
```

This is typical error checking. In the new version of Maple, it could have been done in the procedure declaration as **proc(R:Ring)**.

```
        if nargs = 1 then P := newCategory(); else P := args[2] fi;
        addCategory(P,UnivariatePolynomial);

        if hasCategory(R,Field)
        then P := EuclideanDomain(P)
        ...
        else P := Ring(P)
        fi;

        if hasCategory(R,OrderedSet) then P := OrderedSet(P); fi;
```

Here we have included the usual theorems about univariate polynomials and an example of multiple inheritance. Multiple inheritance in Gauss means that more than one set of operations is added to the domain.

```
        defOperation( CoefficientRing, P &-> Ring, P );
        P[CoefficientRing] := R;
        ...
```

There are lots of operations defined for polynomials of course. One difference between AXIOM and Gauss is that programs in Gauss can get their hands on parts of domains, their parameters e.g. here the coefficient ring.

```
        env := ['D' = P, 'C' = R];
        ...
        P[EuclideanNorm] := subs(env, proc(a) D[Degree](a) end);
```

This is a Maple problem. We are in the process of building the domain P. Here we are inserting the Maple procedure for the EuclideanNorm operation. It needs to call the operation Degree from the domain P. The reason for the reference to `D[Degree]` instead of `P[Degree]` is because Maple does not support nested scopes, and the substitution is a just hack to make this work. Nested scopes will be supported in a future version of Maple. We mention this because apart from having to package call every operation, which is not really serious, this is the only dissatisfying feature of the readability of the code.

```
    ...
    op(P);
  end;
```

Finally, let's look at a domain which inherits the definitions and code from the category UnivariatePolynomial.

```
  DenseUnivariatePolynomial := proc() local x,P,R,env;

    R := args[1];
    P := UnivariatePolynomial(R);
```

Inherit the definitions and code from the category

```
    ...
    if hasProperty(R,CanonicalForm) then
        addProperty(P,CanonicalForm);
        if hasProperty(R,UniquelyRepresented) then
                addProperty(P,UniquelyRepresented);
                P['='] := <evalb(x=y)>;
        fi;
    fi;
```

This shows that Gauss keeps some other kind of information around called properties. For categories, we have properties associated with the mathematical properties of the category like "commutativity". For domains, we have properties associated with the data representation, like "CanonicalForm" and "UniquelyRepresented". Here we are exploiting the fact that if we have a canonical form for the coefficients, then because we are using a dense expanded representation, we have a canonical form for the polynomial ring we are constructing. Secondly, if the coefficients are also uniquely represented in memory, then because we are using a Maple list, the polynomials will be uniquely represented. And why is this information useful? Because now we can use a faster implementation of equality, based on machine address – which is what this Maple code is doing.

```
    # Representation is a Maple list of coefficients
    P[0] := [R[0]];
    P[1] := [R[1]];
    ...
    op(P)
  end:
```

3 Signature Functions in Gauss

The signature functions that we use, in general, map values into finite rings, namely the integers mod n where n is not necessarily prime, and $\mathbb{Z}_p[x]/(m(x))$ where $m(x)$ is not necessarily irreducible, and p is a prime integer. These finite rings are chosen so that there are a low percentage of un-invertible elements. To simplify the presentation of this section however, we assume finite fields.
Our first idea is that domains for which we can define signature functions should support the operation

$$\text{ModularMapping: } (N, N) \rightarrow \text{UNION(} (F\text{:FiniteField, } D \rightarrow F\text{), FAIL)}$$

where N is the set of natural numbers, D is the domain for which this operation is being defined. Thus ModularMapping outputs a pair, a finite field F, and a mapping from D to F. Since for probabilistic algorithms, we will need more than one such mapping, ModularMapping takes an index as the first argument and hence can be used to generate a sequence of mappings. The second argument is a lower bound for the cardinality of the finite field. If D is itself a finite field, then the only non-trivial mapping available is the identity mapping. For this reason, `ModularMapping` is allowed to fail.

The operation ModularMapping has been explicitly coded for the Integer domain and the FiniteField category. In all other cases, it is automatically constructed by the system when a domain is constructed. The constructions for polynomials and quotient fields are simple. We include the code for polynomials here. Note, we have removed the magic substitutions (which are used for simulating nested scopes) from the Maple code to make it easier to read. I.e. the reader should assume that Maple has nested scopes.

```
PolynomialHomomorphism := proc(n,p) local R,F,f,alpha,D,x;
    R := P[CoefficientRing];
    F := R[ModularHomomorphism](n,p);
    if F = FAIL then RETURN(FAIL) fi;
    f := F[1]; F := F[2];
    beta := F[Random]();
    D := DUP(F,x);
    proc(x) local b;
            b := map(f,P[ListCoeffs](x));
            b := D[Polynom](b);
            D[Eval](b,beta);
    end, F
end:
```

In [9] we have looked at several possible constructions for algebraic number fields. Here we sketch a general method for algebraic extensions. Given the algebraic extension $\mathbb{K}[x]/(a)$, we first obtain a modular mapping f into a finite field $\mathbb{F}_q$ for the ground field $\mathbb{K}$ such that the leading coefficient of a does not vanish under f. Then we map the defining polynomial a into $\mathbb{F}_q[x]$ yielding a'. Since $\mathbb{F}_q$ is a finite field, we factor a' and choose to work with the smallest factor b and hence construct the finite field $\mathbb{F}_q[x]/(b)$ in which to compute signatures.

We want to show how we can implement Schwarz's algorithm using this ModularMapping function. Before we can do this, we need some way to determine a degree bound for the degree of the determinant. One way to do this is to implement a special routine. But this would mean coding a new version of the determinant routine. Another way of computing a bound is to simply count the number of multiplications m done in the coefficient field and note that 2^m is a bound on the degree. This is not very good bound but it can be easily implemented. A better method is given in the next section. Assuming we have a degree bound D we can sketch the implementation.

```
# Input: an m by n matrix A over R, a degree bound D,
#        and an error tolerance E
# Output: rank(A) with probability > 1-E of being correct
ProbabilisticMatrixRank := proc(M,A,D,E)
local R,m,n,i,F,N,f,B,rank,error;
    R := M[CoefficientRing]; m := M[Rows](A); n := M[Cols](A);
    error := 1;
    rank := 0;

    for i while error > E do
        F := R[ModularHomomorphism](i,D);
        if F = FAIL then RETURN(FAIL) fi;
        f := F[1]; F := F[2];
        N := Matrix(m,n,F); # Create an m by n matrix domain over F
        B := N[Input](M[Output](N[Map](f,A)));
        rank := max(rank,N[Rank](B));
        error := error * D/F[Size];
    od;
    rank
end;
```

Other applications of ModularMapping include testing if a polynomial divides another, computing the degree of the GCD of two univariate polynomials, hence testing whether two polynomials are relatively prime or not. A good application of the latter is in determining whether a univariate polynomial is square free or not, i.e. whether $\mathrm{GCD}(a, a') = 1$. The coding of these applications is straightforward.

4 Applications

We give several other applications of signatures in this section.

4.1 Signature domains and forms of expressions

The **Signature** domain computes with signatures for rational functions in a set of variables over a constant field in which we can compute signatures. The **Signature** domain simply replaces input expressions by signatures; the ensuing computation computes over the finite ring defined by the signature function. This eliminates intermediate expression swell and provides us with probabilistic zero equivalence but restricts us to the field operations $+, -, \times$, and $/$.

When given as a parameter to a domain constructor, such as DenseUnivariatePolynomial or SquareMatrix, we get a computation in which part of the computation is being done with signatures, and another part is being done with variables. This allows one to look at the *structure* or *form* of the result. This is best understood by looking at an example.

```
> S := Signature(a,b,c):
> R := RationalFunction(S,[x,y,z]):
```

We are computing with rational expressions in 6 symbols. But the symbols a, b, c will be mapped to a finite field on input, so the actual computation will be done with x, y, z only.

```
> M := SM(3,R):  # Create a 3 by 3 matrix domain over R
> A := [[a,b,c],[a*x,a*y,a*z],[a*x^2,a*y^2,a*z^2]]:
> M[Output](M[Inv](M[Input](A)));
                     2       2      2       2
                  * y z  + * y  z  * z  + * y    * z + * y
                [[--------------, -----------, ---------],
                        %1            %1          %1

                       2       2      2       2
                    * x z  + * x  z  * z  + * x    * z + * x
                   [--------------, -----------, ---------],
                          %1            %1          %1

                       2       2      2       2
                    * x y  + * x  y  * y  + * x    * y + * x
                   [--------------, -----------, ---------]]
                          %1            %1          %1

                           2       2      2                2     2
   %1 :=          (y + * x) z  + (* y  + * x ) z + * x y  + * x  y
```

The *'s appearing in the output are signatures which are neither 0 nor 1. Thus the output tells us what the inverse of the matrix looks like as a function of x, y, z by telling us which coefficients are 0 and also which are 1. Note, one could also output the coefficients which are small integers, e.g. -1 might also be of interest.

4.2 Computation Sequences and Automatic Code Generation

The ComputationSequence domain creates a "computation sequence" i.e. a sequence of all the field operations done during a calculation, but without doing the operations symbolically. This gives us a third possibility for automatic code generation. For example, consider the problem of generating efficient numerical code for inverting a given matrix A where the entries of A are a function of $x_1, x_2, \ldots, x_n$. There are three possible approaches:

1. The numeric approach: Here the values of the parameters are bound before the matrix is constructed. The resulting matrix is purely numerical, and a numerical solver is used.

2. The symbolic approach. Here, the parameters are bound after the inverse is computed symbolically. The problem with this approach, is that the symbolic inverse may have no compact representation.
3. The computation sequence approach. Here, the sequence of operations that would be performed in the numeric approach is created in advance. Signatures are used to determine intermediate zeroes so that non-zero pivots are selected.

We illustrate the idea with an example.

```
> S := ComputationSequence(a,b,c):
> M := SM(3,S):
> A := [[a,b,c],[a,b,a],[a,c,b]]:
> B := M[Input](A):
> M[Det](B);
t1 = 1/a
t2 = a-c
t3 = c-b
t4 = b-c
t5 = -a
t6 = t5*t3
t7 = t6*t2
```

We have output the computation sequence as a sequence of assignment statements on the fly, that is, as each arithmetic operation is executed. The t variables are temporary variables. The last variable t7 is the determinant.

4.3 A Probabilistic Method for Computing the Degree of a Polynomial

We give an algorithm that given a polynomial represented by a computation sequence, such as the result of the above determinant calculation, determines probabilistically the degree of each variable. The method was also used in the DAGWOOD system [5]. This is quite an important utility because it means that we can now remove the need to compute degree bounds from many calculations. For example, this can be used to improve the performance of the sparse polynomial interpolation by determining the actual degrees of the polynomials rather than using a bound.

The idea is to create a computation sequence f for the algorithm, then to compute the degree d of it by trying to interpolate the polynomial. After having evaluated f at i points, and interpolated those i points, we evaluate the interpolated polynomial at a next point x_{i+1} yielding y_{i+1} and compare with $f(x_{i+1})$. If the values agree, then we output i as the "probable" degree of the polynomial. If not, we iterate.

To do this efficiently, one needs an incremental version of a polynomial interpolation algorithm that interpolates the polynomial for each successive point in a linear number of operations in F. We have adapted algorithm 5.2 "Newton Interpolation Algorithm" from [2] for this purpose.

Procedure Probabilistic Polynomial Degree Bound

Input: f a computation sequence for a polynomial in n variables over F a finite ring, and k the index of the variable for which the degree is sought

Output: the degree of the kth variable of f

```
for $i = 1..n$ do choose $a_i$ at random from $F$
choose $x_0$ at random from $F$
$y_0 \leftarrow f(a_1, \ldots, a_{k-1}, x_0, a_{k+1}, \ldots a_n)$
for $i = 1..\infty$ do
    $g \leftarrow$ interpolate $x_1, \ldots, x_i$ and $y_1, \ldots, y_i$ over $F$ incrementally
    choose $t$ at random from $F$ such that $t$ is not in $x_1, \ldots, x_i'$
    $y_{i+1} \leftarrow f(a_1, \ldots, a_{k-1}, t, a_{k+1}, \ldots, a_n)$
    if $g(a_1, \ldots, a_{k-1}, t, a_{k+1}, \ldots, a_n) = y_{i+1}$ then output $i$
    $x_{i+1} \leftarrow t$
```

Note this assumes that the cardinality of F is greater than the degree, and that the computation sequence f is polynomial in x_k, otherwise, the algorithm will not terminate. Also, if the computation sequence includes divisions, the algorithm may fail due to an unlucky division by zero. A practical implementation must allow for these cases.

5 Conclusion

We have designed a system that supports parameterized domains, a la AXIOM, in which signature functions are automatically created for many integral domains. This makes it possible to use probabilistic methods to do various calculations much faster than is otherwise possible. We have shown some examples of how signature functions can be used to solve various kinds of problems, in addition to the standard applications.

We expect that the reader will have questions about Gauss so the remainder of this conclusion gives some general information about Gauss. The main advantage of Gauss in Maple is that it allows us to implement generic algorithms, in an AXIOM-like manner. The primary advantages are that it is very simple, very flexible, and very small. We have found that programmers have had no difficulty in writing code.

Is Gauss efficient? Yes and no. Gauss code is written in Maple, thus interpreted. The overhead of Gauss, in comparison with Maple code, consists of a Maple table subscript and procedure call for every operation. The subscript is cheap, the procedure calling mechanism in Maple is relatively expensive. But in many cases, we can directly make use of builtin Maple functions. We find that Gauss runs typically not much slower than the Maple interpreter. For polynomial arithmetic over $\mathbb{Z}, \mathbb{Q}, \mathbb{Z}_p$, Gauss is much slower than Maple because $+, -, \times$ and division are builtin. What we have done to get back the efficiency, is to make Maple polynomial domains in Gauss which use the Maple representation for polynomials, and hence also Maple's builtin operations. On the other hand, in some cases, Gauss has turned out to be much faster than Maple because no time is wasted analyzing what kind of expression was input.

Is Gauss code aesthetically pleasing to read and easy to write? An obvious disadvantage of Gauss is that one must explicitly "package call" each operation. There is no compiler, so no type analysis to avoid having to do this. However, it has been our experience that this is of little or no hindrance to programming in Gauss. The drawback is in interactive usage. The other main deficiency is that since Maple does not support nested scopes, this has to be simulated which makes the code look somewhat ugly. We intend to resolve this deficiency by adding nested scoping to Maple.

References

1. Jenks R., Sutor R.: *axiom - The Scientific Computation System*, Springer, 1992.
2. Geddes K.O., Labahn G., Czapor S.R.: *Algorithms for Computer Algebra* Kluwer, 1991.
3. Gonnet G.H.: Determining Equivalence of Expressions in Random Polynomial Time. it Proceedings of the 16th ACM Symposium on the Theory of Computing (1984) 334–341
4. Gonnet G.H.: New Results for Random Determination of Equivalence of Expressions. *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation* (1986) 127–131
5. Freeman T., Imirzian G., Kaltofen, E.: DAGWOOD: A System for Manipulating Polynomials Given by Straight-Line Programs. *Proceedings of the 1986 Symposium on Symbolic and Algebraic Computation* (1986) 169–175
6. Char B.W., Geddes K.O., Gonnet G.H., Leong B.L., Monagan M.B., and Watt S.M.: *Maple V Language Reference Manual.* Springer-Verlag, New York, 1991.
7. Rabin, M.O.: Probabilistic Algorithm for Testing Primality. *J. of Number Theory* **12** (1980) 128–138
8. Solavay, R. Strassen, V.: A fast Monte-Carlo Test for Primality. *SIAM J. of Computing* **6** (1977) 84–85
9. Monagan M.B.: Signatures + Abstract Types = Computer Algebra - Intermediate Expression Swell. Ph.D. Thesis, University of Waterloo, 1989.
10. Monagan M.B.: A Heuristic Irreducibility Test for Univariate Polynomials *J. Symbolic Comp.* **13** No. 1 (1992) 47–57
11. Schwartz J.T.: Fast probabilistic algorithms for verification of polynomial identities. *J. ACM.* **27** (1980) 701–717