In-place Arithmetic for Polynomials over $\mathbf{Z}_n$

Michael Monagan

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

Abstract. We present space and time efficient algorithms for univariate polynomial arithmetic operations over **Z** mod n where the modulus n does not necessarily fit into is not a machine word. These algorithms provide the key tools for the efficient implementation of polynomial resultant gcd and factorization computation over **Z**, without having to write large amounts of code in a systems implementation language.

1 Background

This paper reports a solution to a dilema we faced during the design and implementation of certain crucial operations in the Maple system [15] namely, computing polynomial resultants, greatest common divisors and factorization in $\mathbf{Z}[x]$.

The efficient implementation of polynomial greatest common divisors (Gcds) is perhaps the most single important part of a general purpose computer algebra system. Gcd computation is the bottleneck of many operations. This is because any calculations which involve rational operations will require Gcd computations in order to reduce fractions to lowest terms. For example, in solving a system of equations with polynomial coefficients, polynomial Gcd calculations will be needed to simplify the solutions. Whereas we can use Euclids algorithm to compute integer Gcds relatively efficiently, compared with integer multiplication and division, the efficient computation of polynomial Gcds is much more difficult. And although the classical algorithms for multiplying and dividing polynomials are fine for most practical calculations, the use of Euclidean based Gcd algorithms results in a phenomenon known as "intermediate expression swell" which causes many intermediate calculations to "blow up". Research on Gcd computations in the 1970's and 1980's [9, 19, 21, 7, 12] led to efficient algorithms for the computation of polynomial Gcds. However, the efficient implementation of these algorithms is quite difficult. The modular based algorithms require efficient computation over the integers mod n. Efficiency is lost in a systems implementation language like C and Lisp, and even more so in an interpreted language.

The efficient implementation of polynomial resultants is not as important as polynomial Gcds. And it has become less important since one of the main applications of resultants, namely in solving systems of polynomial equations, has largely been superceded with the Grobner basis approach [5, 10]. However, it has been our experience that some hard problems can only be solved by clever application of resultants [16]. Thus it is still useful to have an efficient implementation of polynomial resultants. The most efficient algorithms for resultants for dense polynomials are those based on modular methods [9, 4].

Polynomial factorization is another important facility in a computer algebra system. Various algorithms require polynomial factorization, such as computing with algebraic numbers. But also, factorization is one way the user can try to "simplify" an expression. Factored polynomials are usually smaller in size than expanded polynomials.

In 1986, in Maple version 4.2, these operations were implemented as follows. Resultants were computed using the sub-resultant algorithm [9, 4]. Gcds were computed using the Heuristic algorithm GCDHEU [7]. And polynomial factorization over $\mathbf{Z}$ was done using the Berlekamp-Hensel procedure [13, 10].

For dense polynomials, it had been known for some time that the modular methods of Collins [9] were superior to the sub-resultant algorithm for polynomial Resultant and Gcd computation. However, an efficient implementation requires an efficient implementation of polynomial arithmetic in $\mathbf{Z}_p[x]$ where p will be a machine word size prime, and an efficient implementation of the Chinese remainder algorithm for combining images. Maple did not use this approach because firstly, the implementation in Maple would be completely interpreted, and secondly, Maples implementation of the sub-resultant algorithm for computing Resultants and the GCDHEU algorithm for Gcds was quite competitive for many problems. Essentially, Maple was able to "piggy back" off its efficient implementation of multi-precision integer arithmetic and univariate polynomial arithmetic over $\mathbf{Z}$. But the implementation of the Berlekamp-Hensel procedure for factorization requires polynomial arithmetic and linear algebra over $\mathbf{Z}_p$ and polynomial arithmetic over $\mathbf{Z}_{p^k}$. This was all implemented in Maple. Maple was very slow at factoring over finite fields and consequently also slow at factoring over $\mathbf{Z}$. For some time this was not seem as a serious problem because the competition, namely REDUCE and MACSYMA, also had and still have relatively slow univariate polynomial factorization packages. However, just how slow Maple was, and REDUCE and MACSYMA for that matter, became apparent when SMP timings were reported. To give one comparison, the landmark SIGSAM problem No. 7 [11], which includes a factorizaton of a polynomial of degree 40 with over 40 digit coefficients could be done in about a minute on a Vax 11/780 using SMP but took Maple well over an hour.

So, how do we improve the performance of factorization in $\mathbf{Z}[x]$ in Maple? We recognize that Maple is too slow because it is interpreted. Thus what we have to do is to implement the polynomial factorization package and the tools it requires in C. There will be some economy of code since many of the same tools needed for factorization can also used to implement Gcds and resultants efficiently. However, anyone who has implemented a polynomial factorization package knows that this is a formidable task. The amount of coding is considerable. For instance, it is said that when Arthur Normal first implemented polynomial factorization in REDUCE, the overall size of the whole system doubled! I have seen the the SMP implementation. I do not know how many lines of code it was but it was a stack of paper about 1 inch think. This solution presents us with a major dilema in Maple. We have for years tried to avoid coding in C and instead to code in the Maple programming language. The advantage is clear. Maple code is easier to write, read, debug and maintain. We also wanted to keep the Maple kernel, that is, the part of the Maple system that is written in C, as small as possible. This was done primarily so that there would be more storage available for data. However, there are other clear advantages in

maintenance and portability. But, Maple is interpreted and hence some operations will execute slowly. Operations with small integers and floating point numbers excute particularly slowly in Maple compared with compiled C. Hence also polynomial arithmetic and linear algebra over $\mathbf{Z}_n$ executes slowly. One is tempted to simply code all these operations in C. Thus the dilema we faced was, how do we get an efficient polynomial factorization package without writing tens of thousands of lines of C code.

To summarize our finding here, the answer appears to be that one can have an efficient implementation of polynomial resultants, Gcds and factorizaton over $\mathbf{Z}$ if the system supports efficient arithmetic in $\mathbf{Z}_n[x]$. It is not necessary to code everything in C. Specifically, only addition, subtraction, multiplication, division, quotient, remainder, evaluation, interpolation Gcd and resultant in $\mathbf{Z}_n[x]$ need to be coded in C. Factorization over $\mathbf{Z}_p$, and resultants, Gcds and factorization over $\mathbf{Z}$ can then be coded efficiently in terms of these primitives.

Our other major finding is that if we focus on coding these primitives for $\mathbf{Z}_n[x]$ carefully, in particular, multiplication quotient and remainder, we can get modest improvements of factors of 3 to 5 by being careful about the way we handle storage and reduce modulo $\mathbf{Z}_n$. The result is a package which is partially written in C and mostly written in Maple. That amount of C code that we wrote is 1200 lines. This investment gives us fast implementations of the three key operations mentioned previously, and also factorization over $\mathbf{Z}_\mathbf{p}$.

Since then we have used these primitives to improve the performance of other operations. We have implemented the multiple modular method of Collins [9] for bivariate polynomial resultants and Gcds. Also we have used the fast arithmetic for $\mathbf{Z}_p[x]$ to represent large finite fields $\mathrm{GF}(p^k) = \mathbf{Z}_p[x]/(a)$ where $a \in \mathbf{Z}_p[x]$ is irreducible. One then obtains a fast implementation of univariate polynomial arithmetic, including Gcds, resultants and factorization over $\mathrm{GF}(p^k)$.

We note that the computationally intensive steps of the two fast Gcd algorithms compared by Smedley [18] for computing Gcds of univariate polynomials over an algebraic number field which is a simple algebraic extension of $\mathbf{Q}$ also use our codes. The first method [17] is a heuristic method that reduces the problem to a single Gcd computation over $\mathbf{Z}_n$ for a large possibly composite modulus n. The second method [14] is a multiple modular method. Gcds are computed over $\mathbf{Z}_{p_i}[x]/(a)$ for word size prime moduli p_i and combined by application of the Chinese remainder theorem.

2 Introduction

How does one implement efficient univariate polynomial arithmetic over $\mathbf{Z}$, in particular the operations Gcd, resultant, and factorization? The fastest known practical method for computing Gcds and resultants is the dense modular method. For a full description of this method we refer the reader to [9, 10]. Briefly, given $a, b \in \mathbf{Z}[x]$, one computes the $\mathrm{Gcd}(a, b)$ (the $\mathrm{resultant}(a, b)$) modulo primes $p_1, \ldots, p_n$ and then combines the images using the Chinese remainder theorem. There are many details but this is the basic idea. In order to do this most efficiently, one chooses the primes $p_1, \ldots, p_n$ to be the biggest primes that fit into a machine word, so that one can use machine arithmetic directly for calculations in $\mathbf{Z}_{p_i}$. One also needs an efficient implementation of Chinese remaindering for combining the image Gcds or resultants.

This can be implemented efficiently by representing the polynomials over $\mathbf{Z}_{p_i}$ as arrays of machine integers and using the Euclidean algorithm for computing the Gcd and resultant. However, an implementation in C will lose efficiency because in order to multiply in $\mathbf{Z}_p$ with remainder, one can only use half a machine word as otherwise the product will overflow and the leading bits will be lost. Even though almost all hardware has instructions for multiplying two full word numbers and getting the two word result, these instructions are not accessible from C. Lisp implementations lose efficiency because their representation of integers is special. Some bits may be used for special purposes and there is an overhead for arithmetic operations. Note, in AXIOM there is the additional overhead of function calls. Basically, for various reasons, machine efficiency is lost.

Note: if one wants to handle multivariate polynomials, one will also need efficient implementations in C of polynomial evaluation and interpolation. The implementation of the Gcd and resultant computation over $\mathbf{Z}$ can be implemented in the high level language, and even if interpreted, the overhead be relatively insignificant.

Polynomial factorization is considerably more difficult. Given an efficient implementation of polynomial addition, multiplication, quotient and remainder, Gcd, over $\mathbf{Z}_p$ one can write an efficient procedure for factorization of univariate polynomials over $\mathbf{Z}_p$ using the Cantor-Zassenhaus distinct degree factorization algorithm [6]. The bottleneck of this computation is computing the remainder of a^n divided b for large n where $a, b \in \mathbf{Z}_p[x]$. This can be done efficiently using binary powering with remainder and requires only multiplication and remainder operations in $\mathbf{Z}_p[x]$. Note, the efficiency of this procedure can be improved slightly if one can square a polynomial efficiencly. This is worth doing. We found that it saves about 15% overall. The next part of the Berlekamp-Hensel procedure for factorization is to lift the image factors using P-adic lifting (Hensel lifting) from $\mathbf{Z}_p$ to $\mathbf{Z}_{p^k}$ until p^k bounds twice the largest coefficient that could appear in any factor over $\mathbf{Z}$. The details of Hensel lifting can be found in [10, 13]. From our view in this paper, what this means is that one must be able to do polynomial arithmetic over $\mathbf{Z}_{p^k}$ efficiently. In particular, multiplication, quotient and remainder. During the lifting, the modulus p^i will eventually exceed the word size and one is forced to use multi-precision arithmetic. How can we efficiently multiply and divide over $\mathbf{Z}_{p^k}$? The next section of this paper addresses this problem. The factorization problem is then completed by trying combinations of the lifted factors to see if they divide the original input. Again, the details are many. Good references include the texts [10, 13].

3 In Place Algorithms

In this section we design an efficient environment for computing with univariate polynomials over the finite rings $\mathbf{Z}_n$ where n is too large to fit in a machine word, and $\mathbf{Z}_p[x]/(a)$ where $a \in \mathbf{Z}_p[x]$ and p here is word size prime. Let R denote either of these finite rings. Our implementation uses classical algorithms for arithmetic in R, since, in almost all cases, the size of the rings will not be large enough to warrant the use of asymptotically fast algorithms. Our implementation also uses classical algorithms for $R[x]$, since again, for most cases, the degree of the polynomials will not be large enough for asymptotically fast algorithms to win out. We are going to optimize the implementation at the level of storage management and data representation.

In a *generic* implementation of univariate polynomial arithmetic over R (as one would find in AXIOM for example) each arithmetic operation in R implies the creation of a new objects. Each new object created means storage management overhead. For example, to multiply a, b in $\mathbf{Z}_n$ we would first compute $c = a \times b$ then the result c mod n. In doing so several pieces of storage will be allocated which will later have to be garbage collected. We describe a more efficient strategy for computing in $R[x]$ which eliminates this overhead. The idea is to exploit the fact that unlike arithmetic over an arbitrary ring e.g. $\mathbf{Q}$, the storage required for arithmetic operations over R is bounded a priori since R is a finite ring.

We exploit this by coding arithmetic to run *in-place*. For arithmetic operations in $R[x]$ we will either pre-allocate the storage needed for the entire operation or, write out the answer as we go using an *on-line* algorithm. The algorithms given for $R[x]$ all allocate linear total storage in the size of the inputs, assuming the inputs are dense, which they usually are. In the case of polynomial multiplication and division, we can do this in the space required to write down the answer plus a constant number of scratch registers for arithmetic in R. Thus our algorithms are space optimal up to lower order terms. Another significant improvement can be obtained by allowing values to accumulate before reducing modulo n (or a) hence eliminating expensive operations.

The assumption here is that storage management; that is the overhead of allocating storage for each operation, and garbage collection is significant compared with the arithmetic operations involved. The overhead of storage management is surely the main reason why numerical software systems are inherently faster than symbolic algebra systems. This is simply because the primitive objects being manipulated in numerical software systems, namely floating point numbers, have fixed size. Because of this, immediate storage structures can be used for vectors and matrices of floating point numbers. Storage management is trivial in comparison. Now the size of objects in $\mathbf{Z}_n$ and $\mathbf{Z}_p[x]/(a)$ is not fixed. It is parameterized by n, p and $deg(a)$. However, unlike $\mathbf{Q}$, the size depends only on the domain, not on the values of the domain. The difference is significant. For operations over Q the size of the result will depend on additional parameters such as the degree of a polynomial. Although it may be possible to bound the storage needed for arithmetic over $\mathbf{Q}$ and design in-place algorithms, it is so much more difficult that we consider it to be pointless.

3.1 In-place Multiplication

Let $a, b \in R[x]$. The algorithm for in-place polynomial multiplication (IUPMUL) computes the product $c = ab$ using the Cauchy product rule

$$c_k = \sum_{\max(0,k-db) \le i \le \min(k,da)} a_i b_{k-i} \quad \text{for } k = 0..da + kb$$

where $da = deg(a)$ and $db = deg(b)$. The reason for this choice over the more familiar iteration

$$c_{i+j} = a_i b_j \quad \text{for } i = 0..da \text{ for } j = 0..db$$

is that we can sequentially write down the product with additional space for only two scratch registers. The size of the scratch registers depends on R and is given below.

Algorithm IUPMUL: In-place Univariate Polynomial Multiplication

```
IPUPMUL( (a,b,c):Array R, (da,db):Z, (t1,t2):R, m:R ): Z
 - Inputs: univariate polynomials a, b over R of degree da and db
- working storage registers t1, t2 and space for the product in c
- and the modulus m in R
 - Outputs: degree of the product and the product in c
     if da=-1 or db=-1 then return -1
     dc := da + db
     for k in 0 .. dc repeat
          copyinto(0_R,t1)
          for i in max(0,k-db) .. min(k,da) repeat
               InPlaceMul(a[i],b[k-i],t2)
               InPlaceAdd(t2,t1,t1)
          InPlaceRem(t1,m)
          copyinto(t1,c[k])
     - compute the degree of the product
     while dc ≥ 0 and c(dc) = 0_R repeat dc := dc - 1
     return dc
```

Note that the algorithm allows values to accumulate hence removing the remainder operation from the inner loop which often saves over half the work in practice. In the case of $\mathbf{Z}_n$ integer division is relatively expensive for small n compared with integer multiplication. In Maple, integer multiplication is about 3 times faster than integer division.

We also implemented a non-in-place version of the Karatsuba multiplication algorithm for $\mathbf{Z}_n$ for comparison. Note that the break even point will depend on the size of n as well as the degree of the polynomials. For a 20 digit prime we found the break even point to be around degree 64 indicating that IUPMUL is indeed quite efficient.

Implementation Notes We assume the following functions in R. The utility operation copyinto(x, y) copies the value pointed to by x into the space pointed to by y. The function InPlaceMul(x, y, z) computes the product xy in the space pointed to by z. Likewise InPlaceAdd(x, y, z) and InPlaceSub(x, y, z) compute the sum and difference respectively of x and y in the space pointed to by z. The function InPlaceRem(x, y) computes the remainder of x divided y in the space pointed to by x. A note about the representation.

The arrays a, b, c are arrays of pointers to pieces of storage which must be large enough to hold the largest possible values in R. When R is $\mathbf{Z}_n$ the temporaries $t1, t2$ need to be large enough to be able to accumulate at most $min(da, db) + 1$ integers

of magnitude at most $(n-1)^2$. When R is $\mathbf{Z}_p[x]/(a)$ where a is a polynomial of degree $k > 0$, p is a word size prime modulus and elements of R are represented as dense arrays of coefficients, then, the temporaries $t1, t2$ need to be able to store a polynomial of degree $2k-2$ hence $2k-1$ words.

3.2 In-place Division with Remainder

In the in-place algorithm for polynomial division over R we again employ an on-line algorithm to compute the coefficients of first the quotient q then the remainder r of a divided b requiring additional space for two scratch registers. As was the case for multiplication we can remove the remainder operation from the inner loop allowing values to accumulate.

Algorithm IUPDIV: In-place Univariate Polynomial Division

```
IUPDIV( (a,b):Array R, (da,db):Z, (t1,t2):R, (lb,m):R ): Z ==
 - Inputs: univariate polynomials a, b ≠ 0 over R of degree da and db,
- working storage t1, t2, the inverse lb of the leading coefficient of b,
- and the modulus m in R
 - Outputs: the degree of the remainder dr where the quotient of a
- divided b is in a[da − dq..da] and the remainder is in a[0..dr]
    if da < db then return da
    dq := da-db
    dr := db-1
    for k in da..0 by -1 repeat
        copyinto(a[k],t1)
        for j in max(0,k-dq)..min(dr,k) repeat
            InPlaceMul(b[j],a[k-j+db],t2)
            InPlaceSub(t1,t2,t1)
        InPlaceRem(t1,m)
        if t1 < 0_R then InPlaceAdd(m,t1,t1)
        if k ≥ db then
            InPlaceMul(lb,t1,t1)
            InPlaceRem(t1,m)
        copyinto(t1,a[k])
    - now compute the degree of the remainder
    while dr ≥ 0 and a[dr] = 0 repeat dr := dr - 1
    return dr
```

An additional advantage of this on-line algorithm is that if one only needs the quotient then the algorithm (modified to count from da down to the db computes the quotient without computing the remainder hence saving half the work for the case $da = 2db$.

3.3 In-place Gcd

The functionality of algorithm IUPDIV yields a simple in-place algorithm (IUPGCD) for computing Gcd's over R. Note: algorithm IUPGCD returns an unnormalized Gcd.

Algorithm IUPGCD: In-place Univariate Polynomial GCD

IUPGCD((a,b):Array R, (da,db):$\mathbf{Z}$, (t1,t2,t3,t4):R, m:R): (Array R, $\mathbf{Z}$)
- Inputs: univariate polynomials a and b over R of degree da and db
- additional working storage $t1, t2, t3, t4$ and modulus m in R
- Outputs: the degree of the Gcd and a or b which contains the Gcd

```
if da < db return IUPGCD(b,a,db,da,t1,t2,t3,t4,m)
if db = -1 return(b,db)
while db ≥ 0 repeat
    copyinto(b[db],t3)
    copyinto(m,t1)
    InvInPlace_R(t3,t1,t2,t4) - t3 contains the inverse
    dr := IUPDIV(a,b,da,db,t1,t2,t3,m)
    (a,b) := (b,a) - interchange pointers only
    da := db
    db := dr
return(a,da)
```

In this case additional scratch registers are needed by InvInPlace to compute the inverse (if it exists) of an element of R using the half extended Euclidean algorithm see [10] in-place. That is given a, b in R we solve $sa + tm = g$ for s. If $g = 1$ then s is the inverse of a modulo m. We have implemented the half extended Euclidean algorithm for the Euclidean domains $\mathbf{Z}$ and $\mathbf{Z}_p[x]$ where p is a word size prime in-place. Note also with slight modifications, algorithm IUPGCD can be extended to compute univariate polynomial resultants over R.

4 The modp1 Function in Maple

In this section we give further details about the Maple implementation. for arithmetic in $\mathbf{Z}_n[x]$. Note, we have not implemented the case $R = \mathbf{Z}_p[x]/(a)$ internally. Our first attempt at implementing fast arithmetic in $\mathbf{Z}_p[x]$ began with the idea that we should write the key functions (multiplication, quotient and remainder, Gcd and resultants) in the *mod* package in C. That is, the data representation for $\mathbf{Z}_n[x]$ would be a Maple general sum of products data structure. The interface would be via the *mod* function and the coding would require conversions from the Maple representation to an internal dense array representation. However, it became clear early on that the conversion overhead, was very expensive. Or, correctly put, the real work being done could be done very fast. Even for polynomials of degree 100, the time spent converting took much longer than any of multiplication, quotient and remainder, or Gcds.

The *modp*1 function in Maple does univariate polynomial arithmetic over $\mathbf{Z}_n$ using special data representations. Modp1 handles both the case of a word size modulus n separately from the case where the modulus n is large. The case of $n = 2$ is also treated specially. The actual data representation used depends on the size of n. If

$$n < \text{prevprime } \lfloor \sqrt{MAXINT} \rfloor$$

where MAXINT is the largest positive integer representable by the hardware, e.g. on 32 bit machines using signed integers, MAXINT $= 2^{31} - 1$, then a polynomial is represented internally as a dense array of machine integers. Classical algorithms are used with tricks to speed up various cases. For example, for the case $n = 2$ bit operations are used. For otherwise a small modulus additions in polynomial multiplication and division are allowed to accumulate if they cannot overflow. Note the prime here is used for a random number generator. If the modulus n is greater then this number, the a polynomial is represented as a dense array of pointers to Maple integers (multi-precision integers). And the in-place algorithms described in the previous section are used. A example of usage for a large modulus follows

```
> p := prevprime(10^10);
                                        p := 9999999967

> a := modp1( Randpoly(4), p );

        a := [1110694326, 3633074819, 4256145114, 8458720791, 7419670467]

# This represents the polynomial
> modp1( ConvertOut(a,x), p );
                   4                  3                  2
     7419670467 x  + 8458720791 x  + 4256145114 x  + 3633074819 x + 1110694326

> b := modp1( Randpoly(4), p );

        b := [2062222184, 2974124144, 4305615901, 5580039851, 6753832980]

> g := modp1( Randprime(4), p );

            g := [4685305298, 2712797428, 1717237881, 3687530853, 1]

> ag := modp1( Multiply(a,g), p ):
> bg := modp1( Multiply(b,g), p ):
> modp1( Gcd(ag,bg), p );

                [4685305298, 2712797428, 1717237881, 3687530853, 1]

> modp1( Factors(ag), p );

 [7419670467, [[[3203615647, 1], 1], [[7211058641, 1284247953, 9477941733, 1], 1],

                    [[4685305298, 2712797428, 1717237881, 3687530853, 1], 1]] ]
```

Where note the output format of the Factors function is

$$[u, [[f_1, e_1], \ldots [f_n, e_n]]] \;=\; u \times f_1^{e_1} \times \ldots \times f_n^{e_n}$$

Note, the **mod** function in Maple provides a nicer interface for the user to these facilities. For example, in the following, the Maple polynomials will be automatically converted into the *modp1* representation where the computation is done, then

converted back on output. The *evalgf1* package uses the *modp1* facility to implement efficient univariate polynomial arithmetic over finite rings and fields $\mathbf{Z}_p[x]/(a)$. Polynomials are represented as dense arrays of *modp1* polynomials. This facility is also accessed via the mod function with conversions taking place automatically. For example

```
> f := x^8+x^4+x^3+x+1;
                                    8    4    3
                              f := x  + x  + x  + x + 1

> Factor(f) mod 2;
                                     8    4    3
                                    x  + x  + x  + x + 1

> alias(a=RootOf(f,x)):
# Factor f over GF(2^8) = Z2[x]/(f)
> Factor(f,a) mod 2;

      6    3    2           6    4    3    2                    7    6    5    2
(x + a  + a  + a  + 1) (x + a  + a  + a  + a  + a) (x + a) (x + a  + a  + a  + a )

          2         4         4    3                  7    6    5    4    3
    (x + a ) (x + a ) (x + a  + a  + a + 1) (x + a  + a  + a  + a  + a  + a)

> Expand(") mod 2;
                                     8    4    3
                                    x  + x  + x  + x + 1
```

We make some comments about the efficiency gain compared with Maple 4.2 where most of these operations were interpreted. Polynomial multiplication over $\mathbf{Z}_n$ is about 15-30 times faster for a small modulus and polynomial quotient and remainder, Gcd and resultant, and factorization, are all $100 - 400$ times faster. For the case of a large modulus, the improvement is typically a factor of $3 - 15$ where the larger the modulus, the less the improvement. We will not present here any timing comparisons with other systems. We refer the reader to [20] for timing comparisons for polynomial factorizations over $\mathbf{Z}_p$ and $\mathbf{Z}$. We do mention that Maple V can compute the resultant in SIGSAM problem #7 [11] in 26 seconds on a Vax 11/85 and factor it in a further 26 seconds.

We list here a summary of which functions in the *modp1* package we have coded in C and which we have coded in Maple. Note this depends on whether the modulus is small or large. And that all operations coded in C run in linear space.

Operation	Small	Large	Description
Degree	Y	Y	
Ldegree	N	N	The degree of the first non-zero term
Coeff	Y	Y	The coefficient of x^i
Diff	Y	N	The derivative
Shift	Y	Y	Shift a polynomial by x^n where $n \in \mathbf{Z}$
Add	Y	Y	+
Sub	Y	Y	−
Multiply	Y	Y	×
Power	N	N	^
Rem	Y	Y	Polynomial remainder (optionally computes the quotient)
Quo	Y	Y	Polynomial quotient (optionally computes the remainder)
Divide	N	N	Polynomial exact division
Gcd	Y	Y	Polynomial greatest common divisor
Resultant	Y	N	Polynomial resultant
Gcdex	Y	N	The extended Euclidean algorithm
Eval	Y	Y	Polynomial evaluation
Interp	Y	N	Polynomial interpolation
Powmod	Y	N	Compute Rem(a^n,b) using binary powering
Randpoly	Y	N	Generate a polynomial with random coefficients
Sqrfree	N	N	Square-free factorization
Roots	N	N	Compute the roots of a polynomial
Irreduc	N	N	Irreducibility test
Factors	N	N	Factorization (Cantor-Zassenhaus distinct degree)

5 Conclusion

We found that by implementing the primitive operations addition and subtraction, multiplication, quotient and remainder, Gcd and resultant, evaluation and interpolation in $\mathbf{Z}_n[x]$ in C, one is able to then write efficient code for factorization over $\mathbf{Z}_p$ and polynomial resultants, Gcds, and factorization over $\mathbf{Z}$ in a high level language. Even if the high level language is interpreted, the efficiency lost is negligible because the bulk of the work done is the primitive operations. This investment in systems code can also be used to implement several other important algorithms efficiently. Firstly, Collins modular method for Gcds and resultants [9] of dense multivariate polynomials over $\mathbf{Z}$ which we have implemented for bivariate polynomials in Maple. Secondly, efficient polynomial arithmetic over finite fields and rings given by $\mathbf{Z}_p[x]/(a) \ : \ a \in \mathbf{Z}_p[x]$ including resultants, Gcds and factorization. Thirdly, efficient univariate polynomial Gcd computation over a simple algebraic extension of $\mathbf{Q}$, [17, 18, 14].

We also found that the careful use of in-place arithmetic for the key operations multiplication, quotient and remainder over $\mathbf{Z}_n$ results in a significant overall efficiency gain. The gains made by using in-place arithmetic are first that we are able to essentially eliminate the overhead of storage management. Second, we save operations by allowing values to accumulate temporarily. Our implementation in Maple typically results in improvements of factors of 3 to 5 depending on the size of the modulus n.

If computer algebra systems are to get the most efficiency out of the systems hardware for basic arithmetic domains $\mathbf{Z}$, $\mathbf{Z}_n$ and $\mathbf{Z}_p[x]/(a)$, then we believe that this will happen only given careful attention to the functionality provided so that we can build polynomial, vector and matrix arithmetic over these domains without interaction with storage management at every operation. This issue of storage management is even more acute in parallel systems.

Finally, what about linear algebra over $\mathbf{Z}_n$? Although we have not considered vector and matrix operations, it is not difficult to imagine similar schemes whereby one is able to perform vector and matrix arithmetic, such as determinants, in-place. We expect that one would see comparable improvements in performance. Thus it would seem that the next thing to do is to implement a similar facility to the *modp1* facility for vector and linear algebra where the key routines will be matrix multiplication and Gaussian elimination.

References

1. Berlekamp E.R. Factoring Polynomials over Finite Fields. *Bell System Technical Journal*, No 46 1853-1859, 1967.
2. Berlekamp E.R. Factoring Polynomials over Large Finite Fields. *Mathematics of Computation*, **24** 713-715, 1970.
3. Brown W.S. On Euclid's Algorithm and the Computation of Polynomial Greatest Common Divisors. *JACM*, **18**, 478-504, 1971.
4. Brown W.S., Traub J.F. On Euclid's Algorithm and the Theory of Subresultants. *JACM*, **18**, 505-514, 1971.
5. B. Buchberger, A Theoretical Basis for the Reduction of Polynomials to Canonical Forms, ACM SIGSAM Bulletin, **9**, (4), November 1976.
6. Cantor D.G., Zassenhaus H. A New Algorithm for Factoring Polynomials over a Finite Field. *Mathematics of Computation*, **36**, 587-592, 1981.
7. Char, B.W., Geddes K.O., Gonnet, G.H. GCDHEU: Heuristic Polynomial GCD Algorithm Based On Integer GCD Computation. *Proceedings of Eurosam 84*. Springer-Verlag Lecture Notes in Computer Science, **174**, 285-296, 1984.
8. Collins G.E. Subresultants and Reduced Polynomial Remainder Sequences. *JACM*, **14**, 128-142, 1967.
9. Collins G.E. The Calculation of Multivariate Polynomial Resultants. *JACM*, **18**, 515-532, 1971.
10. Geddes K.O., Labahn G., Czapor S.R. *Algorithms for Computer Algebra*. To appear 1992.
11. Johnson S.C., Graham R.L. Problem #7 SIGSAM Bulletin issue number 29, **8** (1), February 1974.
12. Kaltofen E. Computing with Polynomials Given by Straight-Line Programs: I Greatest Common Divisors. *Proceedings of the 17th Annual ACM Symposium on the Theory of Computing*, 131-142, 1985.
13. Knuth, D.E. *The Art of Computer Programming* Vol. 2: *Seminumerical Algorithms* (Second Edition). Addison-Wesley, Reading Massachusetts, 1981.
14. Langemyr, L., McCallum, S. The Computation of Polynomial Greatest Common Divisors. *Proceedings of EUROCAL*, 1987.
15. Maple V Language Reference Manual Springer-Verlag, 1991.

16. R. Peikert, D. Wuertz, M. Monagan, C. de Groot Packing Circles in a Square: A Review and New Results. IPS Research Report No. 91-17, September 1991 ETH-Zentrum, CH-8092 Zurich, Switzerland.
17. Geddes K.O., Gonnet G.H., Smedley T.J. Heuristic Methods for Operations with Algebraic Numbers. *Proceedings of the ACM-SIGSAM 1988 International Symposium on Symbolic and Algebraic Computation*, ISSAC '88, 1988.
18. Smedley T.J. A New Modular Algorithm for Computation of Algebraic Number Polynomial Gcds. *Proceedings of the ACM-SIGSAM 1988 International Symposium on Symbolic and Algebraic Computation*, ISSAC '89, 91-94, 1989.
19. David Yun, The Hensel Lemma in Algebraic Manipulation. Ph.D. Thesis, Massachusetts Institute of Technology.
20. Paul Zimmerman, A comparison of Maple V, Mathematica 1.2 and Macsyma 3.09 on a Sun 3/60. mathPAD newsletter 1 (2), April 1991. Gruppe mathPAD, Universität Paderborn.
21. Zippel R. Probabilistic Algorithms for Sparse Polynomials. *Proceedings of Eurosam 79.* Springer-Verlag Lecture Notes in Computer Science, No 72, 216-226, 1979.