Implementing sparse rational function interpolation in Maple with application to solving parametric linear systems.

Michael Monagan, Mantej Sokhi, and Archit Srivastava

Department of Mathematics, Simon Fraser University, Burnaby, BC V5A 1S6, Canada
mmonagan@sfu.ca, mantej_sokhi@sfu.ca and archit_srivastava@sfu.ca

Abstract. We have experimented with using Kaltofen and Yang's sparse rational function interpolation algorithm to compute the rational function solutions of n by n linear systems in m parameters. We have implemented the algorithm in Maple with some subroutines coded in C for greater efficiency. Our paper describes the algorithm, our implementation, the efficiency of Maple's foreign function interface, and it compares this approach with Lipson's fraction-free algorithm.

Keywords: Rational Function Interpolation, Parametric Linear Systems, Black-box Algorithms

1 Introduction

Let $\mathbb{F}$ be a field and $h \in \mathbb{F}(y_1, y_2, \ldots, y_m)$. So h is a rational function in m variables $y_1, y_2, \ldots, y_m$. Let $\mathbf{B}$ be a black box for h, that is, a computer program that, given a point $\alpha \in \mathbb{F}^m$, $\mathbf{B}(\alpha)$ computes $h(\alpha)$. The Kaltofen-Yang algorithm from [6] interpolates $h(y_1, y_2, \ldots, y_m)$ from values of h computed using $\mathbf{B}$. In this work we have implemented the Kaltofen-Yang algorithm in Maple with parts of it coded in C, and we apply it to solve parametric linear systems over $\mathbb{F} = \mathbb{Q}$.

A square parametric linear system $Ax = b$ over $\mathbb{Q}$ is a system of n equations in n unknowns $x_1, x_2, \ldots, x_n$ where the entries of the matrix A and vector b are polynomials in m parameters $y_1, y_2, \ldots, y_m$. So if $\det(A) \neq 0$ then the solutions are, in general, rational functions in $\mathbb{Q}(y_1, y_2, \ldots, y_m)$. For example, for

$$A = \begin{bmatrix} y_1 & y_2 & y_3 \\ y_2 & y_1 & y_2 \\ y_3 & y_2 & y_1 \end{bmatrix} \qquad b = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

we have $\det(A) = (y_1 - y_3)\left(y_1^2 + y_1 y_3 - 2y_2^2\right)$ and the solutions are

$$x_1 = \frac{y_1 - y_2}{y_1^2 + y_1 y_3 - 2y_2^2} \quad x_2 = \frac{y_1 - 2y_2 + y_3}{y_1^2 + y_1 y_3 - 2y_2^2} \quad x_3 = \frac{y_1 - y_2}{y_1^2 + y_1 y_3 - 2y_2^2}.$$

Computer Algebra systems like Maple and Magma can solve such systems. They use variants of fraction-free Gaussian elimination such as Lipson's algorithm [8].

If $A^{(i)}$ is the n by n matrix constructed by replacing column i of A with b, Cramer's rule says the solutions of $Ax = b$ are given by

$$x_i = \frac{\det(A^{(i)})}{\det(A)} \quad \text{for} \quad 1 \le i \le n.$$

Lipson's algorithm computes $\det(A)$ then $\det(A^{(i)})$ using $O(n^3)$ ring operations in $\mathbb{Q}[y_1, y_2, \ldots, y_m]$ and $O(n^3)$ exact divisions in $\mathbb{Q}[y_1, y_2, \ldots, y_m]$. It also computes intermediate polynomials which are larger than $\det(A)$ and $\det(A^{(i)})$. In a final step Lipson's algorithm simplifies the fraction $\det(A^{(i)})/\det(A)$ by computing and dividing out by $h_i = \gcd(\det(A^{(i)}), \det(A))$.

For many linear systems all $h_i = 1$ and no simplification occurs. For some linear systems, $h_i \neq 1$ and the simplified solutions are much smaller than the polynomials $\det(A^{(i)})$ and $\det(A)$. We propose to apply Kaltofen-Yang to interpolate the simplified solutions directly which should be faster in such cases.

Our paper is organized as follows. In Section 2 we recall details of the Kaltofen-Yang sparse rational function interpolation algorithm from [6]. In Section 3 we describe our implementation of it in Maple. Because Maple is an interpreted language, we have implemented some subroutines in C for greater efficiency. We use Maple's foreign function to access our C subroutines. We investigate the overhead of the foreign function interface. In Section 4 we compare Lipson's algorithm with our implementation of Kaltofen-Yang on a family of parametric linear systems. Our software is available on the web at
`httpd://www.cecm.sfu.ca/~mmonagan/code/KYsolve`

2 The Kaltofen-Yang Algorithm

Let $\mathbb{F}$ be a field and let h be a rational function in $\mathbb{F}(y_1, y_2, \ldots, y_m)$. Thus $h = f/g$ for some polynomials $f, g \in \mathbb{F}[y_1, y_2, \ldots, y_m]$ with $\gcd(f, g) = 1$. The polynomials f, g are unique up to a scalar $\mu \in \mathbb{F}$ since $f/g = (\mu f)/(\mu g)$. Let $\mathbf{B}$ be a black box for computing h. The Kaltofen-Yang algorithm [6] interpolates μf and μg from points computed using $\mathbf{B}$ for some $\mu \in \mathbb{F}$.

In our application $\mathbb{F} = \mathbb{Q}$. If we were to run Kaltofen-Yang over $\mathbb{Q}$, large rationals numbers will be created. To avoid this we use a modular version of Kaltofen-Yang, that is, we interpolate μf and μg modulo a sequence of primes $p = p_1, p_2, \ldots, p_k$ and then we use Chinese remaindering and rational number reconstruction [14] to recover the rational coefficients in μf and μg. So from now on we assume $\mathbb{F} = \mathbb{F}_p$ with p a large prime.

Kaltofen-Yang first pick $\beta_2, \ldots, \beta_m$ from $[1, p-1]$ at random. Suppose we want to compute $f(\sigma)$ and $g(\sigma)$ for some point $\sigma \in \mathbb{F}_p^m$. Kaltofen-Yang interpolate the rational function

$$T(x) = \frac{f(x, \beta_2(x-\sigma_1)+\sigma_2, \ldots, \beta_m(x-\sigma_1)+\sigma_m)}{g(x, \beta_2(x-\sigma_1)+\sigma_2, \ldots, \beta_m(x-\sigma_1)+\sigma_m)}. \tag{1}$$

Thus $T(x) = r(x)/t(x)$ for polynomials $r(x)$ and $t(x)$ in $\mathbb{F}_p[x]$. Observe that

$$T(\sigma_1) = \frac{f(\sigma_1, \sigma_2, \ldots, \sigma_m)}{g(\sigma_1, \sigma_2, \ldots, \sigma_m)} = h(\sigma).$$

Thus $r(\sigma_1) = \mu f(\sigma)$ and $t(\sigma_1) = \mu g(\sigma)$ for some scalar $\mu \in \mathbb{F}_p$.

If we know $\deg(f)$ and $\deg(g)$ we can interpolate $T(x)$ with $D = \deg(f) + \deg(g) + 1$ points[1]. We pick $\alpha_1, \alpha_2, \ldots, \alpha_D$ from $\mathbb{F}_p$ at random, and compute

$$z_i = \mathbf{B}(\alpha_i, \beta_2(\alpha_i - \sigma_1) + \sigma_2, \ldots, \beta_m(\alpha_i - \sigma_1) + \sigma_m) = T(\alpha_i). \tag{2}$$

Next we interpolate (α_i, z_i) to obtain $u(x) \in \mathbb{F}_p[x]$ such that $u(\alpha_i) = z_i$ and compute $m(x) = \prod_{i=1}^{D}(x - \alpha_i)$. Finally (see Section 5.7 of [4]) we solve $r(x)/t(x) \equiv u(x) \mod m(x)$ for $r(x)$ and monic $t(x)$ using the Euclidean algorithm.

The $\beta's$ serve two purposes. Let $g = \sum_{i=0}^{d} g_i$ where each term of g_i has degree i, for example,

$$g = (x_2 - x_3)x_1^2 + 3x_1x_2 + 5x_3^2 + 7 = \underbrace{(x_2x_1^2 - x_3x_1^2)}_{g_3} + \underbrace{(3x_1x_2 + 5x_3^2)}_{g_2} + \underbrace{7}_{g_0}.$$

Notice that if $\beta_2 = 1, \beta_3 = 1$ and $\sigma = (1, 1, 1)$ then $g(x, \beta_2(x - \sigma_1) + \sigma_2, \beta_3(x - \sigma_1) + \sigma_3) = 0x^3 + 8x^2 + 7$ has degree 2 instead of 3. By choosing the $\beta's$ randomly from $[1, p-1]$, we have

$$\Pr[g_d(x, \beta_2, \ldots, \beta_m) = 0] \leq \frac{\deg(g_d)}{(p-1)} = \frac{\deg(g)}{(p-1)}$$

by the Schwartz-Zippel lemma. So the first purpose of the $\beta's$ is to prevent a degree loss with high probability. Notice also that if

$$\begin{aligned} q(x) &= g(x, \beta_2(x - \sigma_1) + \sigma_2, \ldots, \beta_m(x - \sigma_1) + \sigma_m) \\ &= g_d(1, \beta_2, \ldots, \beta_m)x^d + \text{lower degree terms}, \end{aligned}$$

the leading coefficient of $q(x)$ depends on β only, and not on σ. So if $T(x) = r(x)/t(x)$ with $t(x)$ monic, then $t(x) = \mu g(x, \beta_2(x-\sigma_1)+\sigma_2, \ldots, \beta_m(x-\sigma_1)+\sigma_m)$ where $\mu = g_d(1, \beta_2, \ldots, \beta_m)^{-1}$.

2.1 Ben-Or and Tiwari sparse polynomial interpolation

To interpolate the numerator μf and denominator μg we use the Ben-Or/Tiwari sparse polynomial interpolation algorithm from [1] with $\mathbb{F} = \mathbb{F}_p$. We present the main steps of the Ben-Or/Tiwari algorithm for interpolating μf. The denominator μg is interpolated separately.

Let $\mu f = \sum_{i=1}^{t} a_i M_i(y_1, \ldots, y_m)$ where the coefficients $a_i \in \mathbb{F}_p$ are non-zero and the M_i are monomials. Ben-Or/Tiwari uses the points $\sigma^j = (2^j, 3^j, \ldots, p_n^j)$ for $0 \leq j \leq 2t-1$ where p_n denotes the $n'th$ prime. Let $m_i = M_i(2, 3, \ldots, p_n)$ denote the monomial evaluations and let $\lambda(z) = \prod_{i=1}^{t}(z - m_i)$. For simplicity, assume t is known[2]. The Ben-Or/Tiwari algorithm is:

[1] Khodadad and Monagan [7] show how to use the Euclidean algorithm to determine $\deg(f)$ and $\deg(g)$ with high probability, using $D + 1$ points

[2] Kaltofen and Lee in [5] show how to modify Ben-Or/Tiwari to determine t

1 Compute the values $v_i = \mu f(2^j, 3^j, \ldots, p_n^j)$ for $0 \le j \le 2t-1$. Note, if we are interpolating $h = f/g$ using Kaltofen-Yang, each v_i requires $D = \deg(f) + \deg(g) + 1$ values of $h(y_1, \ldots, y_m)$ for a total of $2tD$ values of h.
2 Compute $\lambda(z)$ from the v_i using the Berlekamp-Massey algorithm [9]. This requires $O(t^2)$ field operations in $\mathbb{F}_p$. We have implemented Berlekamp-Massey in C for $p < 2^{63}$.
3 Factor $\lambda(z)$ over $\mathbb{F}_p$ to determine the monomial evaluations m_i. We use Maple for this. Maple uses the Cantor-Zassenhaus algorithm [3]. The implementation (see [10]) does does $O(t^2 \log p)$ field operations in $\mathbb{F}_p$ and it uses machine arithmetic for $p < 2^{31.5}$.
4 Determine the monomials M_i from m_i by factoring m_i using trial division by $2, 3, 5, \ldots, p_n$. For example, if $m_i = 60 = 2^2 \cdot 3 \cdot 5$ then $M_i = y_1^2\, y_2\, y_3$.
5 Solve for the unknown coefficients a_i of μf. Since $M_i(2^j, 3^j, \ldots, p_n^n) = m_i^j$, we have $v_j = \sum_{i=1}^{t} a_i m_i^j$, a t by t transposed Vandermonde linear system $Va = b$ where $V_{i,j} = m_i^{j-1}$ and $b_i = v_{j-1}$. It can be solved using $O(t^2)$ field operations using Zippel's solver from [15]. We have implemented Zippel's solver in C for $p < 2^{63}$. Our implementation uses $O(t)$ space.

Note, for Step 4 to work we require $p > m_i$. Also, since $\det(V) = \prod_{1 \le i < j \le t}(m_j - m_i)$, the matrix V in Step 5 is non-singular if $p > m_i$. Our Maple implementation is currently limited to 31.5 bit primes and we would like to use 63 bit primes. Consider the monomial $M_i = x_6^d$. We have $m_i = p_6^d = 13^d$. Thus $m_i < 2^{31.5}$ means $d \le 8$ but $m_i < 2^{63}$ means $d \le 17$.

3 Implementing the Kaltofen-Yang algorithm in Maple

We are given an n by n parametric linear system $Ax = b$ over $\mathbb{Q}$ to solve. If we clear fractions we can assume the entries of A and b are polynomials in $\mathbb{Z}[y_1, y_2, \ldots, y_m]$. Since we are using a modular algorithm, we need to solve $Ax = b$ modulo a prime p. We first need to construct a black box $\mathbf{B} : \mathbb{F}_p^m \to \mathbb{F}_p^n$ that takes as input a point $\alpha \in \mathbb{F}_p^m$ and solves $A(\alpha)x = b(\alpha) \mod p$ for $x \in \mathbb{F}_p^n$. We can do this using the following Maple code.

```
MakeBlackBox := proc(A::Matrix,b::Vector,y::list(name)) local m,Ab,B;
   m := nops(y);
   Ab := <A|b>;
   B := proc(alpha::list(integer),p::prime) local S,i,A,det;
      S := {seq( y[i]=alpha[i],i=1..m)};
      A := Eval(Ab,S) mod p;  # A = Ab(alpha) mod p
      A := Matrix(m,m+1,A,datatype=integer[8],order=C_order);
      LinearAlgebra:-Modular:-RowReduce(p,A,m,m+1,m,'det',0,0,0,0,true);
      if det=0 then return FAIL else return A[1..m,m+1]; fi;
   end;
end:
```

The Maple commands `RowReduce` and `Eval` are programmed in C. `RowReduce` only works for 32 bit primes or less. `Eval` works for primes of any size but only

uses machine arithmetic if $p < 2^{32}$. It would be nice if Maple supported primes up to 63 bits here.

We need to interpolate $T(x)$ (see equation (1)) using $D = \deg f + \deg g + 1$ values for each solution $x_i \in \mathbb{F}_p(y_1, y_2, \ldots, y_m)$. Let $T(x) = r(x)/t(x)$ where $\gcd(r,t) = 1$ and t is monic. To compute $r(x)$ and $t(x)$ we first interpolate $u(x) \in \mathbb{F}_p[x]$ such that $u(\alpha_i) = z_i$ (see equation (2)). Maple has a builtin library routine for doing this, namely, `Interp(alpha,z,x) mod p` which is efficient for 31.5 bit primes. We coded Newton interpolation in C for 63 bit primes.

Next we compute $m(x) = \prod_{i=1}^{D}(x - z_i)$ in Maple and then we solve $u(x) \equiv r(x)/t(x) \mod m(x)$ for $r(x)$ and $t(x)$ with $\deg(r) = \deg(f)$ and $\deg(t) = \deg(g)$ and $t(x)$ monic, using the Euclidean algorithm. Again, Maple has a builtin library routine for doing this, namely, `Ratrecon(u,m,x) mod p` which is efficient for 31.5 bit primes. We have also implemented this in C for 63 bit primes.

To call a C subroutine from Maple, we use Maple's foreign function interface (see Chapter 14 of [2]). We use `gcc` to create shared object files. We use one dimensional arrays of 64 bit integers to send polynomials and data from Maple to C and back. For example, our C code for newton.c has this specification.

```
#define LONG long long int
void NewtonInterp(LONG *a, LONG *z, int n, LONG *u, LONG p);
// Compute u(x)=sum(u[i]*x^i,i=0..n-1)) such that u(a[i])=z[i] mod p
```

We compile a shared object file with

```
gcc -O3 -shared -o newton.so -fPIC newton.c
```

The following Maple code returns a Maple procedure which will call our C code.

```
Newton := define_external('NewtonInterp',
   aa::ARRAY(1..nn,datatype=integer[8],
   zz::ARRAY(1..nn,datatype=integer[8],
   nn::integer[4],
   uu::ARRAY(1..nn,datatype=integer[8],
   pp::integer[8],
   LIB="/home/mmonagan/poly/linalg/newton.so");
```

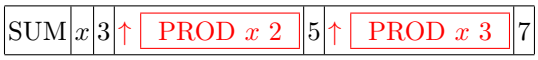
Next we need to call our C code for rational reconstruction to compute $r(x)$ and $t(x)$ such that $r(x)/t(x) \equiv u(x) \mod m(x)$. We first compute $m(x) = \prod_{i=0}^{n-1}(x - a[i])$ in Maple. We then need to convert the Maple polynomial $m(x)$ into an array of coefficients. There is no Maple builtin routine for doing this conversion. The fastest way to do it in Maple is with

```
M := Array(1..n,[seq(coeff(m,x,i),i=0..n)],datatype=integer[8]);
```

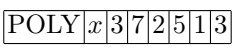
After our rational function reconstruction routine has computed $r(x)$ and $t(x)$ in arrays R and T we need to convert the arrays R and T to Maple polynomials. The fastest way to do this in Maple is to use the `add` command. Assuming $df = \deg(f)$ and $dg = \deg(g)$, we use

```
r := add( R[i]*x^i, i=0..df );
t := add( T[i]*x^i, i=0..dg );
```

The `add` command is implemented in C but the conversion cost is relatively expensive; it can take more time than the C code for computing $r(x)$ and $t(x)$. The cost is illustrated in Table 1. One reason it is slow is because `add` creates intermediate PROD data structures for the monomials x^i. For example, for $R = [0, 3, 5, 7]$, to create $r = 3x + 5x^2 + 7x^3$, Maple first creates the object



then "simplifies" it to Maple's POLY representation [12]



It would be faster if there was a command to create the POLY object directly. We think these conversions should be coded in C so that they are not the bottleneck of the foreign function interface.

Table 1 gives timing data in CPU microseconds for interpolating $u(x)$ using Newton interpolation and solving $u(x) \equiv r(x)/t(x) \mod m(x)$ using the Euclidean algorithm. Column Interp is for Maple's `Interp(...) mod p` command. Column Ratrecon is for Maple's `Ratrecon(...) mod p` command. Columns labeled C are CPU timings for our C subroutines timed in C. Columns labeled FFI include the overhead of the Maple foreign function interface. Column seq is the time for converting the $u(x)$ and $m(x)$ polynomials to arrays. Column add is the time for the two `add` commands.

Table 1. CPU timings in micro seconds using $p = 2^{31} - 1$.

		Newton			Ratrecon			Conversions	
deg f	deg g	Interp	C	FFI	Ratrecon	C	FFI	seq	t add
5	5	20.7	0.76	2.3	38.3	1.10	3.6	18.6	14.4
10	10	25.3	3.08	4.6	54.6	2.44	4.9	29.6	18.6
20	20	39.6	10.3	12.0	93.4	5.45	7.7	53.8	25.7
40	40	89.4	36.0	37.6	188.5	13.4	18.1	108.8	39.7
80	80	270.7	131.9	133.3	444.5	49.2	52.2	207.7	72.9
160	160	949.7	503.4	504.5	1274.7	176.6	177.7	408.9	131.6
320	320	3558.	1951.	1954.	3703.4	658.9	656.3	887.3	236.6

Comparing the C and FFI columns, the Maple foreign function overhead is typically 1 to 2 microseconds which is very good. The overhead of `add` and `seq`, however, is very significant, even at higher degrees.

We end with an optimization. Recall that the solutions $x_i = f_i/g_i$ where $\gcd(f_i, g_i) = 1$ for $1 \le i \le n$. In many parametric linear systems, the denominators g_i will all be equal (up to a scalar). For example, often $g_i = \det(A)$. We can easily identify when this happens and interpolate $\mu g_1(y_1, \ldots, y_m)$ only.

4 Comparison with Lipson's Algorithm

Algorithm FFS below is Lipson's fraction-free algorithm from [8] for solving the n by n linear system $Ax = b$ over the ring $R = \mathbb{Z}[y_1, y_2, \ldots, y_m]$. It is a variation of Gaussian elimination that avoids creating fractions for as long as possible to avoid doing $O(n^3)$ polynomial gcd operations in R.

Algorithm FFS FractionFreeSolve
Input $A \in R^{n \times n}$ and $b \in R^n$ where $R = \mathbb{Z}[y_1, y_2, \ldots, y_m]$
Output $f \in R^n$ and $g \in R^n$ such that $Ax = b$ where $x_i = f_i/g_i$.

```
 1  B := [A|b] // the augmented matrix
 2  μ := 1
 3  for k = 1, 2, ..., n − 1 do
 4      i := k; while i ≤ n and B_{i,k} = 0 do i := i + 1 end while
 5      if i > n then return 0 end if // A is singular
 6      if i > k then interchange row i and k end if
 7      for i = k + 1, k + 2, ..., n do
 8          for j = k + 1, k + 2, ..., n + 1 do
 9              num := B_{k,k} × B_{i,j} − B_{i,k} × B_{k,j}
10              B_{i,j} := num ÷ μ // an exact division in R
11          end for
12          B_{i,k} := 0
13      end for
14      μ := B_{k,k}
15  end for
16  z_n := B_{n,n+1}
17  for i = n − 1, n − 2, ..., 1 do
18      num := B_{i,n+1} × B_{n,n} − Σ_{j=i+1}^{n} B_{i,j} × z_j
19      z_i := num ÷ B_{i,i} // an exact division in R
20  end for
21  for i = 1, 2, ..., n do
22      h_i := gcd(z_i, B_{n,n})
23      f_i := z_i ÷ h_i
24      g_i := B_{n,n} ÷ h_i
25  end for
26  return f, g. // x_i = f_i/g_i
```

Algorithm FFS proceeds in three main steps. The first main step (lines 2 to 15) triangularizes the augmented matrix B using the Bareiss/Edmonds fraction-free Gaussian elimination. The $O(n^3)$ divisions in line 10 by μ are exact in the polynomial ring R. After this step we have $B_{n,n} = \pm \det(A)$. The second main step (lines 16 to 20) computes $z_i = \pm \det(A^{(i)})$ using Lipson's fraction-free back substitution. Again, the $n-1$ divisions in line 19 by $B_{i,i}$ are exact in R. After this step we have $x_i = z_i/B_{n,n} = \det(A^{(i)})/\det(A)$. The third step removes common factors $h_i = \gcd(z_i, B_{n,n})$ from the solutions in lines 22 to 24. These polynomial gcd computations can be expensive when $h_i \neq 1$.

An expression swell occurs in line 9. When $k = n-1, i = n$ and $j = n$ the polynomial $num = \mu B_{n,n} = B_{n-2,n-2} \det(A)$, usually has many more terms

than $\det(A)$. We note that Monagan and Vrbik in [13] used lazy polynomial arithmetic to avoid creating num in expanded form when computing $num \div B_{i,i}$ in line 10. An expression swell also occurs in line 18 where $num = B_{i,i}\, z_i = B_{i,i} \det(A^{(i)})$ a polynomial which usually has many more terms than $\det(A^{(i)})$. The expression swell is measured in Table 2.

We have implemented Lipson's algorithm in Maple. In Maple, all polynomial arithmetic is coded in C. The representation of polynomials is described in [12]. The divisions in lines 10, 19, 23 and 24 use Maple's `divide` command which implements the heap division algorithm of Monagan and Pearce from [11].

4.1 Timing Benchmark

We compare our implementation of Kaltofen-Yang with Lipson's algorithm for $A = Tn$, the symmetric $n \times n$ Toeplitz matrix and $b = [1, 1, \ldots, 1]$ a vector of ones. Here $Tn_{ij} = y_{|i-j|+1}$. The example in the introduction shows T_3 and the solutions of $T_3 x = b$. What is special about these linear systems is that $\det(Tn)$ has two irreducible factors and the denominators g_i of x_i are one of those factors, that is, the other factor cancels out. In the example in the introduction, the factor $y_1 - y_3$ cancels out and $g_i = y_1^2 + y_1 y_3 - 2y_2^2$ for $1 \le i \le 3$.

In Table 2, row $\max \#num$ is the maximum number of terms of the num polynomials in Lipson's algorithm and row Swell is $\max \#num / \max \#g_i$ which measures the expression swell factor in Lipson's algorithm. Row *Lipson* is the time for our Maple implementation of Lipson's algorithm. Rows $\times$, $\div$ and gcd are the time spent in Lipson's algorithm on polynomial multiplication, division, and gcd respectively. Row KY is the time for our implementation of the Kaltofen-Yang algorithm. Rows *interp*, *roots*, and *probe* are the time spent interpolating $T(x)$, computing the computing roots in line 3 of Ben-Or/Tiwari, and solving linear systems modulo primes, respectively. The number of linear systems solved is in row #probes. To generate the timings in Table 2 we used one core of an Intel Xeon 6342 Gold CPU with 128 gigabytes of RAM.

Observe that our implementation of Kaltofen-Yang first beats our Maple implementation of Lipson's algorithm at $n = 8$, and, at $n = 12$, Kaltofen-Yang is 320 times faster than Lispson's algorithm. At $n = 12$ Kaltofen-Yang solved 196,636 linear systems $A(\alpha)x = b(\alpha) \mod p$ which took 6.391 seconds. At $n = 12$ the largest polynomial created by Lipson's algorithm had 82,990,563 terms which is 14,875 times larger than the solutions.

Acknowledgement. This work was supported by NSERC of Canada.

Disclosure of Interests. The authors have no competing interests that are relevant to the content of this article.

References

1. Ben-Or, M., Tiwari, P.: A deterministic algorithm for sparse multivariate polynomial interpolation. In *Proceedings of STOC '88*, pages 301–309, ACM (1988)

Table 2. Timings are in CPU seconds. OM = out of memory. NA = not attempted.

n	6	7	8	9	10	11	12	13	14
$\#\det(Tn)$	120	427	1628	6090	23793	90296	350726	1338076	5165957
$\max \#num$	1512	9206	56007	345008	2122800	13284290	82990563	–	–
$\max \#f_i$	15	52	78	267	430	1623	1623	9757	15872
$\max \#g_i$	32	56	167	294	931	1730	5579	10611	34937
Swell	47	164	335	1173	2280	7679	14875	–	–
Lipson	0.036	0.128	0.728	3.741	53.06	384.5	12678.	OM	NA
$\times$	0.004	0.027	0.157	1.557	15.44	196.7	6425.	–	–
$\div$	0.000	0.001	0.034	0.357	4.45	16.0	1502.	–	–
gcd	0.032	0.100	0.537	1.827	29.11	126.3	4751.	–	–
KY	0.088	0.134	0.498	1.366	3.025	10.05	39.56	181.12	877.1
interp	0.039	0.051	0.261	0.594	1.404	3.638	16.85	42.02	241.5
roots	0.002	0.011	0.026	0.130	0.353	2.308	6.181	59.56	291.3
probe	0.016	0.033	0.098	0.339	0.582	1.419	6.391	16.24	102.6
#probes	768	1052	4124	10268	20508	49180	196636	458780	1835036

2. Bernardin, L., Chin, P., DeMarco, P., Geddes, K.O., Hare, D.E.G., Heal, K.M., Labahn, G., May, J.P., McCarron, J., Monagan, M.B., Ohashi, D., Vorkoetter, S.M.: *Maple Programming Guide*, Maplesoft (2024) `www.maplesoft.com/documentation_center/Maple2024/ProgrammingGuide.pdf`
3. G. Cantor, D.G., Zassenhaus, H.: A new algorithm for factoring polynomials over finite fields. *Mathematics of Computation*, **36**(154):587–592 (1981)
4. von zur Gathen, J., Gerhard, J.: *Modern Computer Algebra.* 3rd edn. Cambridge University Press (2013)
5. Kaltofen, E., Lee, W.: Early termination in sparse interpolation algorithms. *J. Symbolic Computation*, **36**:365–400, Elsevier (2003)
6. Kaltofen, E., Yang, Z.: On exact and approximate interpolation of sparse rational functions. In *Proceedings ISSAC 2007*, pages 203–210, ACM (2007)
7. Khodadad, S., Monagan, M.: Fast rational function reconstruction. In *Proceedings of ISSAC 2006*, pages 184-190, ACM (2006)
8. Lipson, J.D.: Symbolic methods for the computer solution of linear equations with applications to flowgraphs. In *Proceedings of the 1968 Summer Institute on Symbolic Mathematical Computation*, pages 233–303 (1969)
9. Massey, J. L.: Shift-Register Synthesis and BCH Decoding. *IEEE Trans. Information Theory*, **15**:122-127 (1969)
10. Monagan, M.B.: In-place arithmetic for polynomials over $\mathbf{Z}_n$. In *Proceedings of DISCO '92*, LNCS **721**:22–34, Springer (1993)
11. Monagan, M., Pearce, R.: Sparse Polynomial Division Using a Heap. *J. Symbolic Computation*, **46**(7):807–822, Elsevier (2011)
12. Monagan, M., Pearce, R.: The Design of Maple's Sum-of-Products and POLY Data Structures for Representing Mathematical Objects. *Communications in Computer Algebra*, **48**:166–186, ACM (2015)
13. Monagan, M., Vrbik, P. Lazy and Forgetful Polynomial Arithmetic and Applications. *Proceedings of CASC '09*, LNCS **5743**, pp. 226–239, Springer (2009)
14. Wang, P.S., Guy, M.J.T., Davenport, J.H.: P-adic reconstruction of rational numbers. SIGSAM Bulletin, **16**(2):2–3, ACM (1982)
15. Zippel, R.: Interpolating polynomials from their values. *J. Symbolic Computation*, **9**:375–403, Elsevier (1990)