Evaluating parametric holonomic sequences using rectangular splitting

Fredrik Johansson[*]

RISC
Johannes Kepler University
4040 Linz, Austria

fredrik.johansson@risc.jku.at

ABSTRACT

We adapt the rectangular splitting technique of Paterson and Stockmeyer to the problem of evaluating terms in holonomic sequences that depend on a parameter. This approach allows computing the n-th term in a recurrent sequence of suitable type using $O(n^{1/2})$ "expensive" operations at the cost of an increased number of "cheap" operations.

Rectangular splitting has little overhead and can perform better than either naive evaluation or asymptotically faster algorithms for ranges of n encountered in applications. As an example, fast numerical evaluation of the gamma function is investigated. Our work generalizes two previous algorithms of Smith.

Categories and Subject Descriptors

I.1.2 [**Computing Methodologies**]: Symbolic and Algebraic Manipulation—*Algorithms*; F.2.1 [**Theory of Computation**]: Analysis of Algorithms and Problem Complexity—*Numerical Algorithms and Problems*

General Terms

Algorithms

Keywords

Linearly recurrent sequences, Numerical evaluation, Fast arithmetic, Hypergeometric functions, Gamma function

1. INTRODUCTION

A sequence $(c(i))_{i=0}^{\infty}$ is called *holonomic* (or *P-finite*) of order r if it satisfies a linear recurrence equation

$$a_r(i)c(i+r) + a_{r-1}(i)c(i+r-1) + \ldots + a_0(i)c(i) = 0 \quad (1)$$

[*]Supported by the Austrian Science Fund (FWF) grant Y464-N18.

where $a_0, \ldots, a_r$ are polynomials. The class of holonomic sequences enjoys many useful closure properties: for example, holonomic sequences form a ring, and if $c(i)$ is holonomic then so is the sequence of partial sums $s(n) = \sum_{i=0}^{n} c(i)$. A sequence is called *hypergeometric* if it is holonomic of order $r = 1$. The sequence of partial sums of a hypergeometric sequence is holonomic of order (at most) $r = 2$.

Many integer and polynomial sequences of interest in number theory and combinatorics are holonomic, and the power series expansions of many well-known special functions (such as the error function and Bessel functions) are holonomic.

We are interested in efficient algorithms for evaluating an isolated term $c(n)$ in a holonomic sequence when n is large. Section 2 recalls the well-known techniques of rewriting (1) in matrix form and applying *binary splitting*, which gives a near-optimal asymptotic speedup for certain types of coefficients, or *fast multipoint evaluation* which in the general case is the asymptotically fastest known algorithm.

In section 3, we give an algorithm (Algorithm 3) which becomes efficient when the recurrence equation involves an "expensive" parameter (in a sense which is made precise), based on the baby-step giant-step technique of Paterson and Stockmeyer [11] (called *rectangular splitting* in [5]).

Our algorithm can be viewed as a generalization of the method given by Smith in [13] for computing rising factorials. Conceptually, it also generalizes an algorithm given by Smith in [12] for evaluation of hypergeometric series. Our contribution is to recognize that rectangular splitting can be applied systematically to a very general class of sequences, and in an efficient way (we provide a detailed cost analysis, noting that some care is required in the construction of the algorithm to get optimal performance).

The main intended application of rectangular splitting is high-precision numerical evaluation of special functions, where the parameter is a real or complex number (represented by a floating-point approximation), as discussed further in section 4. Although rectangular splitting is asymptotically slower than fast multipoint evaluation, it is competitive in practice. In section 5, we present implementation results comparing several different algorithms for numerical evaluation of the gamma function to very high precision.

2. MATRIX ALGORITHMS

Let R be a commutative ring with unity and of sufficiently large characteristic where necessary. Consider a sequence of rank-r vectors $(c(i) = (c_1(i), \ldots, c_r(i))^T)_{i=0}^{\infty}$ satisfying a recurrence equation of the form

$$\begin{pmatrix} c_1(i+1) \\ \vdots \\ c_r(i+1) \end{pmatrix} = M(i) \begin{pmatrix} c_1(i) \\ \vdots \\ c_r(i) \end{pmatrix} \tag{2}$$

where $M \in R[k]^{r\times r}$ (or $\mathrm{Quot}(R)(k)^{r\times r}$) and where $M(i)$ denotes entrywise evaluation. Given an initial vector $c(0)$, we wish to evaluate the single vector $c(n)$ for some $n > 0$, where we assume that no denominator in M vanishes for $0 \le i < n$. A scalar recurrence of the form (1) can be rewritten as (2) by taking the vector to be $\tilde{c}(i) = (c(i), \ldots, c(i+r-1))^T$ and setting M to the companion matrix

$$M = \frac{1}{a_r} \begin{pmatrix} & a_r & & \\ & & \ddots & \\ & & & a_r \\ -a_0 & -a_1 & \ldots & -a_{r-1} \end{pmatrix}. \tag{3}$$

In either case, we call the sequence holonomic (of order r).

Through repeated application of the recurrence equation, $c(n)$ can be evaluated using $O(r^2 n)$ arithmetic operations (or $O(rn)$ if M is companion) and temporary storage of $O(r)$ values. We call this strategy the *naive algorithm*.

The naive algorithm is not generally optimal for large n. The idea behind faster algorithms is to first compute the matrix product

$$P(0, n) = \prod_{i=0}^{n-1} M(i). \tag{4}$$

and then multiply it by the vector of initial values (matrix multiplication is of course noncommutative, and throughout this paper the notation in (4) is understood to mean $M(n-1)\ldots M(2)M(1)M(0)$). This increases the cost to $O(r^\omega n)$ arithmetic operations where ω is the exponent of matrix multiplication, but we can save time for large n by exploiting the structure of the matrix product. The improvement is most dramatic when all matrix entries are constant, allowing binary exponentiation (with $O(\log n)$ complexity) or diagonalization to be used, although this is a rather special case. We assume in the remainder of this work that r is fixed, and omit $O(r^\omega)$ factors from any complexity estimates.

From this point, we may view the problem as that of evaluating (4) for some $M \in R[k]^{r\times r}$. It is not a restriction to demand that the entries of M are polynomials: if $M = \tilde{M}/q$, we can write $P(0, n) = \tilde{P}(0, n)/Q(0, n)$ where $\tilde{P}(0, n) = \prod_{i=0}^{n-1} q(i)\tilde{M}(i)$ and $Q(0, n) = \prod_{i=0}^{n-1} q(i)$. This reduces the problem to evaluating two denominator-free products, where the second product has order 1.

2.1 Binary splitting

In the *binary splitting* algorithm, we recursively compute a product of square matrices $P(a, b) = \prod_{i=a}^{b-1} M(i)$ (where the entries of M need not necessarily be polynomials of i), as $P(m, b)P(a, m)$ where $m = \lfloor (a + b)/2 \rfloor$. If the entries of partial products grow in size, this scheme balances the sizes of the subproducts in a way that allows us to take advantage of fast multiplication.

For example, take $M(i) \in R[x]^{r\times r}$ where all $M(i)$ have bounded degree. Then $P(a, b)$ has entries in $R[x]$ of degree $O(b - a)$, and binary splitting can be shown to compute $P(0, n)$ using $O(\mathsf{M}(n) \log n)$ operations in R where $\mathsf{M}(n)$ is the complexity of polynomial multiplication, using $O(n)$ extra storage. Over a general ring R, we have $\mathsf{M}(n) = O(n \log^{1+o(1)} n)$ by the result of [6], making the binary splitting softly optimal. This is a significant improvement over the naive algorithm, which in general uses $O(n^2)$ coefficient operations to generate the n-th entry in a holonomic sequence of polynomials.

Analogously, binary splitting reduces the bit complexity for evaluating holonomic sequences over $\mathbb{Z}$ or $\mathbb{Q}$ (or more generally the algebraic numbers) from $O(n^{2+o(1)})$ to $O(n^{1+o(1)})$. For further references and several applications of the binary splitting technique, we refer to Bernstein [2].

2.2 Fast multipoint evaluation

The *fast multipoint evaluation* method is useful when all arithmetic operations are assumed to have uniform cost. Fast multipoint evaluation allows evaluating a polynomial of degree d simultaneously at d points using $O(\mathsf{M}(d) \log d)$ operations and $O(d \log d)$ space. Applied to a polynomial matrix product, we obtain Algorithm 1, which is due to Chudnovsky and Chudnovsky [7].

Algorithm 1 Polynomial matrix product using fast multipoint evaluation

```
Input: $M \in R[k]^{r\times r}$, $n = m \times w$
Output: $\prod_{i=0}^{n-1} M(i)$
1: $[T_0, \ldots, T_{m-1}] \leftarrow [M(k), \ldots, M(k + m - 1)]$
            ▷ Compute entrywise Taylor shifts of the matrix
2: $U \leftarrow \prod_{i=0}^{m-1} T_i$        ▷ Binary splitting in $R[k]^{r\times r}$
3: $[V_0, \ldots, V_{w-1}] \leftarrow [U(0), \ldots, U((w - 1)m)]$
            ▷ Fast multipoint evaluation
4: return $\prod_{i=0}^{w-1} V_i$     ▷ Repeated multiplication in $R^{r\times r}$
```

We assume for simplicity of presentation that n is a multiple of the parameter m (in general, we can take $w = \lfloor n/m \rfloor$ and multiply by the remaining factors naively). Taking $m \sim n^{1/2}$, Algorithm 1 requires $O(\mathsf{M}(n^{1/2}) \log n)$ arithmetic operations in the ring R, using $O(n^{1/2} \log n)$ temporary storage during the fast multipoint evaluation step. Bostan, Gaudry and Schost [4] improve the algorithm to obtain an $O(\mathsf{M}(n^{1/2}))$ operation bound, which is the best available result for evaluating the n-th term of a holonomic sequence over a general ring. Algorithm 1 and some of its applications are studied further by Ziegler [18].

3. RECTANGULAR SPLITTING FOR PARAMETRIC SEQUENCES

We now consider holonomic sequences whose recurrence equation involves coefficients from a commutative ring C with unity as well as an additional, distinguished parameter x. The setting is as in the previous section, but with $R = C[x]$. In other words, we are considering holonomic sequences of polynomials (or, by clearing denominators, rational functions) of the parameter. We make the following definition.

Definition 1. *A holonomic sequence $(c(n) \equiv c(x, n))_{n=0}^{\infty}$ is parametric over C (with parameter x) if it satisfies a linear recurrence equation of the form* (2) *with $M \in R[k]$ where $R = C[x]$.*

Let H be a commutative C-algebra, and let $c(x,k)$ be a parametric holonomic sequence defined by a recurrence matrix $M \in C[x][k]^{r\times r}$ and an initial vector $c(z,0) \in H^r$. Given some $z \in H$ and $n \in \mathbb{N}$, we wish to compute the single vector $c(z,n) \in H^r$ efficiently subject to the assumption that operations in H are expensive compared to operations in C. Accordingly, we distinguish between:

- *Coefficient operations* in C
- *Scalar operations* in H (additions in H and multiplications $C \times H \to H$)
- *Nonscalar multiplications* $H \times H \to H$

For example, the sequence of rising factorials

$$c(x,n) = x^{\overline{n}} = x(x+1)\cdots(x+n-1)$$

is first-order holonomic (hypergeometric) with the defining recurrence equation $c(x,n+1) = (n+x)c(x,n)$, and parametric over $C = \mathbb{Z}$. In some applications, we wish to evaluate $c(z,n)$ for $z \in H$ where $H = \mathbb{R}$ or $H = \mathbb{C}$.

The Paterson-Stockmeyer algorithm [11] solves the problem of evaluating a polynomial $P(x) = \sum_{i=0}^{n-1} p_i x^i$ with $p_i \in C$ at $x = z \in H$ using a reduced number of nonscalar multiplications. The idea is to write the polynomial as a rectangular array

$$\begin{aligned} P(x) = &(p_0 + \ldots + p_{m-1}x^{m-1}) \\ &+ (p_m + \ldots + p_{2m-1}x^{m-1})x^m \\ &+ (p_{2m} + \ldots + p_{3m-1}x^{m-1})x^{2m} \\ &+ \ldots \end{aligned} \tag{5}$$

After computing a table containing $x^2, x^3, \ldots, x^{m-1}$, the inner (rowwise) evaluations can be done using only scalar multiplications, and the outer (columnwise) evaluation with respect to x^m can be done using about n/m nonscalar multiplications. With $m \sim n^{1/2}$, this algorithm requires $O(n^{1/2})$ nonscalar multiplications and $O(n)$ scalar operations.

A straightforward application of the Paterson-Stockmeyer algorithm to evaluate each entry of $\prod_{i=0}^{n-1} M(x,i) \in C[x]^{r\times r}$ yields Algorithm 2 and the corresponding complexity estimate of Theorem 2.

Algorithm 2 Polynomial matrix product and evaluation using rectangular splitting

```
Input: $M \in C[x][k]^{r\times r}$, $z \in H$, $n = m \times w$
Output: $\prod_{i=0}^{n-1} M(z,i) \in H^{r\times r}$
1: $[T_0,\ldots,T_{n-1}] \leftarrow [M(x,0),\ldots,M(x,n-1)]$
       ▷ Evaluate matrix w.r.t. $k$, giving $T_i \in C[x]^{r\times r}$
2: $U \leftarrow \prod_{i=0}^{n-1} T_i$        ▷ Binary splitting in $C[x]^{r\times r}$
3: $V \leftarrow U(z)$
       ▷ Evaluate $U$ entrywise using Paterson-Stockmeyer with step length $m$
4: return $V$
```

Theorem 2. *The n-th entry in a parametric holonomic sequence can be evaluated using $O(n^{1/2})$ nonscalar multiplications, $O(n)$ scalar operations, and $O(\mathsf{M}(n)\log n)$ coefficient operations.*

Proof. We call Algorithm 2 with $m \sim n^{1/2}$. Letting $d = \max \deg_k M$ and $e = \max \deg_x M$, computing $T_0,\ldots,T_{n-1}$ takes $O(nde) = O(n)$ coefficient operations. Since $\deg_x T_i \le e$, generating U using binary splitting costs $O(\mathsf{M}(n)\log n)$ coefficient operations. Each entry in U has degree at most $ne = O(n)$, and can thus be evaluated using $O(n^{1/2})$ nonscalar multiplications and $O(n)$ scalar operations with the Paterson-Stockmeyer algorithm. □

If we only count nonscalar multiplications, Theorem 2 is an asymptotic improvement over fast multipoint evaluation which uses $O(n^{1/2}\log^{2+o(1)} n)$ nonscalar multiplications ($O(n^{1/2}\log^{1+o(1)} n)$ with the improvement of Bostan, Gaudry and Schost).

Algorithm 2 is not ideal in practice since the polynomials in U grow to degree $O(n)$. Their coefficients also grow to $O(n \log n)$ bits when $C = \mathbb{Z}$ (for example, in the case of rising factorials, the coefficients are the Stirling numbers of the first kind $S(n,k)$ which grow to a magnitude between $(n-1)!$ and $n!$). This problem can be mitigated by repeatedly applying Algorithm 2 to successive subproducts $\prod_{i=a}^{a+\tilde{n}} M(z,i)$ where $\tilde{n} \ll n$, but the nonscalar complexity is then no longer the best possible. A better strategy is to apply rectangular splitting to the matrix product itself, leading to Algorithm 3. We can then reach the same operation complexity while only working with polynomials of degree $O(n^{1/2})$, and over $C = \mathbb{Z}$, having coefficients of bit size $O(n^{1/2}\log n)$.

Theorem 3. *For any choice of m, Algorithm 3 requires $O(m + n/m)$ nonscalar multiplications, $O(n)$ scalar operations, and $O((n/m)\mathsf{M}(m)\log m)$ coefficient operations. In particular, the complexity bounds stated in Theorem 2 also hold for Algorithm 3 with $m \sim n^{1/2}$. Moreover, Algorithm 3 only requires storage of $O(m)$ elements of C and H, and if $C = \mathbb{Z}$, the coefficients have bit size $O(m \log m)$.*

Proof. This follows by applying a similar argument as used in the proof of Theorem 2 to the operations in the inner loop of Algorithm 3, noting that U has entries of degree $m \deg_x M = O(m)$ and that the matrix multiplication $S \times V$ requires $O(1)$ nonscalar multiplications and scalar operations (recalling that we consider r fixed). □

Algorithm 3 Improved polynomial matrix product and evaluation using rectangular splitting

```
Input: $M \in C[x][k]^{r\times r}$, $z \in H$, $n = m \times w$
Output: $\prod_{i=0}^{n-1} M(z,i) \in H^{r\times r}$
1: Compute power table $[z^j,\ 0 \le j \le m \deg_x M]$
2: $V \leftarrow 1_{H^{r\times r}}$        ▷ Start with the identity matrix
3: for $i \leftarrow 0$ to $w-1$ do
4:     $[T_0,\ldots,T_{m-1}] \leftarrow [M(x, im+j)]_{j=0}^{m-1}$
           ▷ Evaluate matrix w.r.t. $k$, giving $T_j \in C[x]^{r\times r}$
5:     $U \leftarrow \prod_{j=0}^{m-1} T_j$        ▷ Binary splitting in $C[x]^{r\times r}$
6:     $S \leftarrow U(z)$        ▷ Evaluate w.r.t. $x$ using power table
7:     $V \leftarrow S \times V$        ▷ Multiplication in $H^{r\times r}$
8: return $V$
```

3.1 Variations

Many variations of Algorithm 3 are possible. Instead of using binary splitting directly to compute U, we can generate the bivariate matrix

$$W_m = \prod_{i=0}^{m-1} M(x, k+i) \in C[x][k]^{r\times r} \tag{6}$$

at the start of the algorithm, and then obtain U by evaluating W_m at $k = im$. We may also work with differences of two successive U (for small m, this can introduce cancellation resulting in slightly smaller polynomials or coefficients). Combining both variations, we end up with Algorithm 4 in which we expand and evaluate the bivariate polynomial matrices

$$\Delta_m = \prod_{i=0}^{m-1} M(x, k+m+i) - \prod_{i=0}^{m-1} M(x, k+i) \in C[x][k]^{r \times r}.$$

This version of the rectangular splitting algorithm can be viewed as a generalization of an algorithm used by Smith [13] for computing rising factorials (we consider the case of rising factorials further in Section 5.1). In fact, the author of the present paper first found Algorithm 4 by generalizing Smith's algorithm, and only later discovered Algorithm 3 by "interpolation" between Algorithm 2 and Algorithm 4.

Algorithm 4 Polynomial matrix product and evaluation using rectangular splitting (variation)

```
Input: $M \in C[x][k]^{r \times r}$, $z \in H$, $n = m \times w$
Output: $\prod_{i=0}^{n-1} M(z, i) \in H^{r \times r}$
1: Compute power table $[z^j, 0 \le j \le m \deg_x M]$
2: $\Delta \leftarrow \prod_{i=0}^{m-1} M(x, k+m+i) - \prod_{i=0}^{m-1} M(x, k+i)$
                                   ▷ Binary splitting in $C[x][k]^{r \times r}$
3: $V \leftarrow S \leftarrow \prod_{i=0}^{m-1} M(z, i)$
        ▷ Evaluate w.r.t. $k$, and w.r.t. $x$ using power table
4: for $i \leftarrow 0$ to $w - 2$ do
5:     $S \leftarrow S + \Delta(z, mi)$
        ▷ Evaluate w.r.t. $k$, and w.r.t. $x$ using power table
6:     $V \leftarrow S \times V$
7: return $V$
```

The efficiency of Algorithm 4 is theoretically somewhat worse than that of Algorithm 3. Since $\deg_x W_m = O(m)$ and $\deg_k W_m = O(m)$, W_m has $O(m^2)$ terms (likewise for Δ_m), making the space complexity higher and increasing the number of coefficient operations to $O((n/m)m^2)$ for the evaluations with respect to k. However, this added cost may be negligible in practice. Crucially, when $C = \mathbb{Z}$, the coefficients have similar bit sizes as in Algorithm 3.

Initially generating W_m or Δ_m also adds some cost, but this is cheap compared to the evaluations when n is large enough: binary splitting over $C[x][k]$ costs $O(\mathsf{M}(m^2) \log m)$ coefficient operations by Lemma 8.2 and Corollary 8.28 in [17]. This is essentially the same as the total cost of binary splitting in Algorithm 3 when $m \sim n^{1/2}$.

We also note that a small improvement to Algorithm 3 is possible if $M(x, k+m) = M(x+m, k)$: instead of computing U from scratch using binary splitting in each loop iteration, we can update it using a Taylor shift. At least in sufficiently large characteristic, the Taylor shift can be computed using $O(\mathsf{M}(m))$ coefficient operations with the convolution algorithm of Aho, Steiglitz and Ullman [1], saving a factor $O(\log n)$ in the total number of coefficient operations. In practice, basecase Taylor shift algorithms may also be beneficial (see [16]).

In lucky cases, the polynomial coefficients (in either Algorithm 3 or 4) might satisfy a recurrence relation, allowing them to be generated using $O(n)$ coefficient operations (and avoiding the dependency on polynomial arithmetic).

3.2 Several parameters

The rectangular splitting technique can be generalized to sequences $c(x_1, \ldots, x_v, k)$ depending on several parameters. In Algorithm 3, we simply replace the power table by a v-dimensional array of the possible monomial combinations. Then we have the following result (ignoring coefficient operations).

THEOREM 4. *The n-th entry in a holonomic sequence depending on v parameters can be evaluated with rectangular splitting using $O(m^v + n/m)$ nonscalar multiplications and $O((n/m)m^v)$ scalar multiplications. In particular, taking $m = n^{1/(v+1)}$, $O(n^{1-1/v})$ nonscalar multiplications and $O(n^{2v/(1+v)})$ scalar multiplications suffice.*

PROOF. If $d_i = \deg_{x_i} M \le d$, the entries of a product of m successive shifts of M are C-linear combinations of $x_1^{e_{1,j}} \cdots x_h^{e_{v,j}}$, $0 \le e_{i,j} \le md_i \le md$, so there is a total of $O(m^v)$ powers. $\square$

Unfortunately, this gives rapidly diminishing returns for large v. When $v > 1$, the number of nonscalar multiplications according to Theorem 4 is asymptotically worse than with fast multipoint evaluation, and reducing the number of nonscalar multiplication requires us to perform more than $O(n)$ scalar multiplications, as shown in Table 1. Nevertheless, rectangular splitting could perhaps still be useful in some settings where the cost of nonscalar multiplications is sufficiently large.

v	m	Nonscalar	Scalar
1	$n^{1/2}$	$O(n^{0.5})$	$O(n)$
2	$n^{1/3}$	$O(n^{0.666\ldots})$	$O(n^{1.333\ldots})$
3	$n^{1/4}$	$O(n^{0.75})$	$O(n^{1.5})$
4	$n^{1/5}$	$O(n^{0.8})$	$O(n^{1.6})$

Table 1: Step size m minimizing the number of nonscalar multiplications for rectangular splitting involving v parameters.

4. NUMERICAL EVALUATION

Assume that we want to evaluate $c(x, n)$ where the underlying coefficient ring is $C = \mathbb{Z}$ (or $\overline{\mathbb{Q}}$) and the parameter x is a real or complex number represented by a floating-point approximation with a precision of p bits.

Let $\mathsf{M}_{\mathbb{Z}}(p)$ denote the bit complexity of some algorithm for multiplying two p-bit integers or floating-point numbers. Commonly used algorithms include classical multiplication with $\mathsf{M}_{\mathbb{Z}}(p) = O(p^2)$, Karatsuba multiplication with $\mathsf{M}_{\mathbb{Z}}(p) = O(p^{1.585})$, and Fast Fourier Transform (FFT) based multiplication, such as the Schönhage-Strassen algorithm, with $\mathsf{M}_{\mathbb{Z}}(p) = \tilde{O}(p)$. An unbalanced multiplication where the smaller operand has q bits can be done using $O((p/q)\mathsf{M}_{\mathbb{Z}}(q))$ bit operations [5].

The naive algorithm clearly uses $O(n\mathsf{M}_{\mathbb{Z}}(p))$ bit operations to evaluate $c(x, n)$, or $\tilde{O}(np)$ with FFT multiplication. In Algorithm 3, the nonscalar multiplications cost $O((m + n/m)\mathsf{M}_{\mathbb{Z}}(p))$ bit operations. The coefficient operations cost $\tilde{O}(mn)$ bit operations (assuming the use of fast polynomial arithmetic), which becomes negligible if p grows faster than m. Finally, the scalar multiplications (which are unbalanced) cost

$$O\left(n\,p\,\frac{\mathsf{M}_{\mathbb{Z}}(m\log m)}{m\log m}\right)$$

bit operations. Taking $m \sim n^{\alpha}$ for $0 < \alpha < 1$, we get an asymptotic speedup with classical or Karatsuba multiplication (see Table 2) provided that p grows sufficiently rapidly along with n. With FFT multiplication, the scalar multiplications become as expensive as the nonscalar multiplications, and rectangular therefore does not give an asymptotic improvement.

Mult. algorithm	Scalar multiplications	Naive
Classical	$\tilde{O}(n^{1+\alpha}p)$	$\tilde{O}(np^2)$
Karatsuba	$\tilde{O}(n^{1+0.585\alpha}p)$	$\tilde{O}(np^{1.585})$
FFT	$\tilde{O}(np)$	$\tilde{O}(np)$

Table 2: Bit complexity of scalar multiplications in Algorithm 3 and total bit complexity of the naive algorithm

However, due to the overhead of FFT multiplication, rectangular splitting is still likely to save a constant factor over the naive algorithm. In practice, one does not necessarily get the best performance by choosing $m \approx n^{0.5}$ to minimize the number of nonscalar multiplications alone; the best m has to be determined empirically.

Algorithm 1 is asymptotically faster than the naive algorithm as well as rectangular splitting, with a bit complexity of $\tilde{O}(n^{1/2}p)$. It should be noted that this estimate does not reflect the complexity required to obtain a given *accuracy*. As observed by Köhler and Ziegler [9], fast multipoint evaluation can exhibit poor numerical stability, suggesting that p might have to grow at least as fast as n to get accuracy proportional to p.

When x and all coefficients in M are positive, rectangular splitting introduces no subtractions that can cause catastrophic cancellation, and the reduction of nonscalar multiplications even improves stability compared to the naive algorithm, making $O(\log n)$ guard bits sufficient to reach p-bit accuracy. When sign changes are present, evaluating degree-m polynomials in expanded form can reduce accuracy, typically requiring use of $\tilde{O}(m)$ guard bits. In this case Algorithm 3 is a marked improvement over Algorithm 2.

4.1 Summation of power series

A common situation is that we wish to evaluate a truncated power series

$$f(x) \approx s(x,n) = \sum_{k=0}^{n} c(k)x^k, \quad n = O(p) \tag{7}$$

where $c(k)$ is a holonomic sequence taking rational (or algebraic) values and x is a real or complex number. In this case the Paterson-Stockmeyer algorithm is applicable, but might not give a speedup when applied directly as in Algorithm 2 due to the growth of the coefficients. Since $d(k) = c(k)x^k$ and $s(x,n)$ are holonomic sequences with x as parameter, Algorithm 3 is applicable.

Smith noted in [12] that when $c(k)$ is hypergeometric (Smith considered the Taylor expansions of elementary functions in particular), the Paterson-Stockmeyer technique can be combined with scalar divisions to remove accumulated factors from the coefficients. This keeps all scalars at a size of $O(\log n)$ bits, giving a speedup over naive evaluation when non-FFT multiplication is used (and when scalar divisions are assumed to be roughly as cheap as scalar multiplications). This algorithm is studied in more detail by Brent and Zimmermann [5].

At least conceptually, Algorithm 3 can be viewed as a generalization of Smith's hypergeometric summation algorithm to arbitrary holonomic sequences depending on a parameter (both algorithms can be viewed as means to eliminate repeated content from the associated matrix product). The speedup is not quite as good since we only reduce the coefficients to $O(n^{1/2}\log n)$ bits versus Smith's $O(\log n)$. However, even for hypergeometric series, Algorithm 3 can be slightly faster than Smith's algorithm for small n (e.g. $n \lesssim 100$) since divisions tend to be more expensive than scalar multiplications in implementations.

Algorithm 3 is also more general: for example, we can use it to evaluate the generalized hypergeometric function

$${}_pF_q\left(\begin{matrix}a_1,\ldots,a_p\\ b_1,\ldots,b_q\end{matrix}\,\middle|\, z\right) = \sum_{k=0}^{\infty} \frac{a_1^{\overline{k}}\cdots a_p^{\overline{k}}}{b_1^{\overline{k}}\cdots b_q^{\overline{k}}}\,\frac{w^k}{k!} \tag{8}$$

where a_i, b_i, w (as opposed to w alone) are rational functions of the real or complex parameter x.

An interesting question, which we do not attempt to answer here, is whether there is a larger class of parametric sequences other than hypergeometric sequences and their sums for which we can reduce the number of nonscalar multiplications to $O(n^{1/2})$ while working with coefficients that are strictly smaller than $O(n^{1/2}\log n)$ bits.

4.2 Comparison with asymptotically faster algorithms

If all coefficients in (7) including the parameter x are rational or algebraic numbers and the series converges, $f(x)$ can be evaluated to p-bit precision using $\tilde{O}(p)$ bit operations using binary splitting. An $\tilde{O}(p)$ bit complexity can also be achieved for arbitrary real or complex x by combining binary splitting with translation of the differential equation for $f(x)$. The general version of this algorithm, sometimes called the *bit-burst algorithm*, was developed by Chudnovsky and Chudnovsky and independently with improvements by van der Hoeven [14]. It is used in some form for evaluating elementary functions in several libraries, and a general version has been implemented by Mezzarobba [10].

For high-precision evaluation of elementary functions, binary splitting typically only becomes worthwhile at a precision of several thousand digits, while implementations typically use Smith's algorithm for summation of hypergeometric series at lower precision. We expect that Algorithm 3 can be used in a similar fashion for a wider class of special functions.

When $c(k)$ in (7) involves real or complex numbers, binary splitting no longer gives a speedup. In this case, we can use fast multipoint to evaluate (7) using $\tilde{O}(p^{1.5})$ bit operations (Borwein [3] discusses the application to numerical evaluation of hypergeometric functions). This method does not appear to be widely used in practice, presumably owing to its high overhead and relative implementation difficulty. Although rectangular splitting is not as fast asymptotically, its ease of implementation and low overhead makes it an attractive alternative.

5. HIGH-PRECISION COMPUTATION OF THE GAMMA FUNCTION

In this section, we consider two holonomic sequences depending on a numerical parameter: rising factorials, and the partial sums of a certain hypergeometric series defining the incomplete gamma function. In both cases, our goal is to accelerate numerical evaluation of the gamma function at very high precision.

We have implemented the algorithms in the present section using floating-point ball arithmetic (with rigorous error bounding) as part of the Arb library[1]. All arithmetic in $\mathbb{Z}[x]$ is done via FLINT [8], using a Schönhage-Strassen FFT implemented by W. Hart.

Fast numerically stable multiplication in $\mathbb{R}[x]$ is done by breaking polynomials into segments with similarly-sized coefficients and computing the subproducts exactly in $\mathbb{Z}[x]$ (a simplified version of van der Hoeven's block multiplication algorithm [15]), and asymptotically fast polynomial division is implemented using Newton iteration.

All benchmark results were obtained on a 2.0 GHz Intel Xeon E5-2650 CPU.

5.1 Rising factorials

Rising factorials of a real or complex argument appear when evaluating the gamma function via the asymptotic Stirling series

$$\log \Gamma(x) = \left(x - \frac{1}{2}\right) \log x - x + \frac{\log 2\pi}{2} + \sum_{k=1}^{N-1} \frac{B_{2k}}{2k(2k-1)x^{2k-1}} + R_N(x).$$

To compute $\Gamma(x)$ with p-bit accuracy, we choose a positive integer n such that there is an N for which $|R_N(x+n)| < 2^{-p}$, and then evaluate $\Gamma(x) = \Gamma(x+n)/x^{\overline{n}}$. It is sufficient to choose n such that the real part of $x+n$ is of order βp where $\beta = (2\pi)^{-1} \log 2 \approx 0.11$.

The efficiency of the Stirling series can be improved by choosing n slightly larger than the absolute minimum in order to reduce N. For example, $\operatorname{Re}(x+n) \approx 2\beta p$ is a good choice. A faster rising factorial is doubly advantageous: it speeds up the argument reduction, and making larger n cheap allows us to get away with fewer Bernoulli numbers.

Smith [13] uses the difference of four consecutive terms

$$\begin{aligned}(x+k+4)^{\overline{4}} - (x+k)^{\overline{4}} &= (840 + 632k + 168k^2 + 16k^3) \\ &\quad + (632 + 336k + 48k^2)x \\ &\quad + (168 + 48k)x^2 \\ &\quad + 16x^3\end{aligned}$$

to reduce the number of nonscalar multiplications to compute $x^{\overline{n}}$ from $n-1$ to about $n/4$. This is precisely Algorithm 4 specialized to the sequence of rising factorials and with a fixed step length $m = 4$.

Consider Smith's algorithm with a variable step length m. Using the binomial theorem and some rearrangements, the polynomials can be written down explicitly as

$$\Delta_m = (x+k+m)^{\overline{m}} - (x+k)^{\overline{m}} = \sum_{v=0}^{m-1} x^v \sum_{i=0}^{m-v-1} k^i \, C_m(v,i) \tag{9}$$

where

$$C_m(v,i) = \sum_{j=i+1}^{m-v} m^{j-i} S(m, v+j) \binom{v+j}{v} \binom{j}{i} \tag{10}$$

and where $S(m, v+j)$ denotes an unsigned Stirling number of the first kind. This formula can be used to generate Δ_m efficiently in practice without requiring bivariate polynomial arithmetic. In fact, the coefficients can be generated even cheaper by taking advantage of the recurrence (found by M. Kauers)

$$(v+1)C_m(v+1,i) = (i+1)C_m(v,i+1). \tag{11}$$

We have implemented several algorithm for evaluating the rising factorial of a real or complex number. For tuning parameters, we empirically determined simple formulas that give nearly optimal performance for different combinations of $n, p < 10^5$ (typically within 20% of the speed with the best tuning value found by a brute force search):

- In Algorithm 1, $m = n^{0.5}$.
- Algorithm 2 is applied to subproducts of length $\tilde{n} = \min(2n^{0.5}, 10p^{0.25})$, with $m = \tilde{n}^{0.5}$.
- In Algorithms 3 and 4, $m = \min(0.2p^{0.4}, n^{0.5})$.

Our implementation of Algorithm 4 uses (10) instead of binary splitting, and Algorithm 3 exploits the symmetry of x and k to update the matrix U using Taylor shifts instead of repeated binary splitting.

Figure 5.1 compares the running times where x is a real number with a precision of $p = 4n$ bits. This input corresponds to that used in our Stirling series implementation of the gamma function.

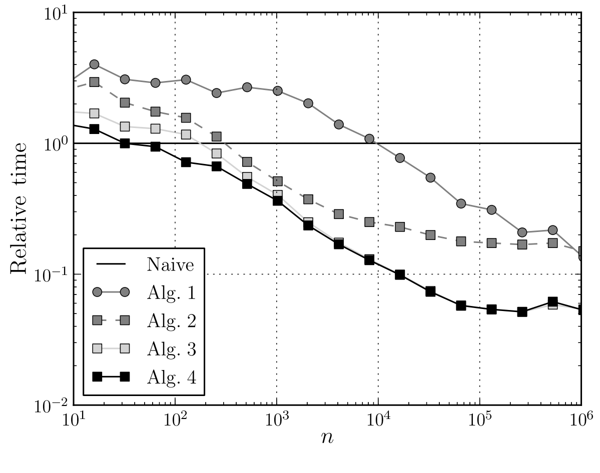

Figure 1: Timings of rising factorial algorithms, normalized against the naive algorithm.

On this benchmark, Algorithms 3 and 4 are the best by far, gaining a 20-fold speedup over the naive algorithm for

[1]`http://fredrikj.net/arb`

large n (the speedup levels off around $n = 10^5$, which is expected since this is the approximate point where FFT integer multiplication kicks in). Algorithm 4 is slightly faster than Algorithm 3 for $n < 10^3$, even narrowly beating the naive algorithm for n as small as $\approx 10^2$.

Algorithm 1 (fast multipoint evaluation) has the most overhead of all algorithms and only overtakes the naive algorithm around $n = 10^4$ (at a precision of $40,000$ bits). Despite its theoretical advantage, it is slower than rectangular splitting up to n exceeding 10^6.

Table 5.1 shows absolute timings for evaluating $\Gamma(x)$ where x is a small real number in Pari/GP 2.5.4, and our implementation in Arb (we omit results for MPFR 3.1.1 and Mathematica 9.0, which both were slower than Pari). Both implementations use the Stirling series, caching the Bernoulli numbers to speed up multiple evaluations. The better speed of Arb for a repeated evaluation (where the Bernoulli numbers are already cached) is mainly due to the use of rectangular splitting to evaluate the rising factorial. The total speedup is smaller than it would be for computing the rising factorial alone since we still have to evaluate the Bernoulli number sum in the Stirling series. The gamma function implementations over $\mathbb{C}$ have similar characteristics.

Decimals	Pari/GP	(first)	Arb	(first)
100	0.000088		0.00010	
300	0.00048		0.00036	
1000	0.0057		0.0025	
3000	0.072	(9.2)	0.021	(0.090)
10000	1.2	(324)	0.25	(1.4)
30000	15	(8697)	2.7	(22)
100000			39	(433)
300000			431	(7131)

Table 3: Timings in seconds for evaluating $\Gamma(x)$ where x is a small real number (timings for the first evaluation, including Bernoulli number generation, is shown in parentheses).

5.2 A one-parameter hypergeometric series

The gamma function can be approximated via the (lower) incomplete gamma function as

$$\Gamma(z) \approx \gamma(z, N) = z^{-1} N^z e^{-N} \, {}_1F_1(1, 1+z, N). \tag{12}$$

Borwein [3] noted that applying fast multipoint evaluation to a suitable truncation of the hypergeometric series in (12) allows evaluating the gamma function of a fixed real or complex argument to p-bit precision using $\widetilde{O}(p^{1.5})$ bit operations, which is the best known result for general z (if z is algebraic, binary splitting evaluation of the same series achieves a complexity of $\widetilde{O}(p)$).

Let $t_k = N^k/(z(z+1)\cdots(z+k))$ and $s_n = \sum_{k=0}^{n} t_k$, giving

$$\lim_{n\to\infty} s_n = {}_1F_1(1, 1+z, N)/z.$$

For $z \in [1, 2]$, choosing $N \approx p \log 2$ and $n \approx (e \log 2)p$ gives an error of order 2^{-p} (it is easy to compute strict bounds). The partial sums satisfy the order-2 recurrence

$$\begin{pmatrix} s_k \\ t_{k+1} \end{pmatrix} = \frac{M(k)}{q(k)} \frac{M(k-1)}{q(k-1)} \cdots \frac{M(0)}{q(0)} \begin{pmatrix} 0 \\ 1/z \end{pmatrix} \tag{13}$$

where

$$M(k) = \begin{pmatrix} 1+k+z & 1+k+z \\ 0 & N \end{pmatrix}, \quad q(k) = 1+k+z. \tag{14}$$

The matrix product (13) may be computed using fast multipoint evaluation or rectangular splitting. We note that the denominators are identical to the top left entries of the numerator matrices, and therefore do not need to be computed separately.

Figure 5.2 compares the performance of the Stirling series (with fast argument reduction using rectangular splitting) and three different implementations of the ${}_1F_1$ series (naive summation, fast multipoint evaluation, and rectangular splitting using Algorithm 3 with $m = 0.2n^{0.4}$) for evaluating $\Gamma(x)$ where x is a real argument close to unity.

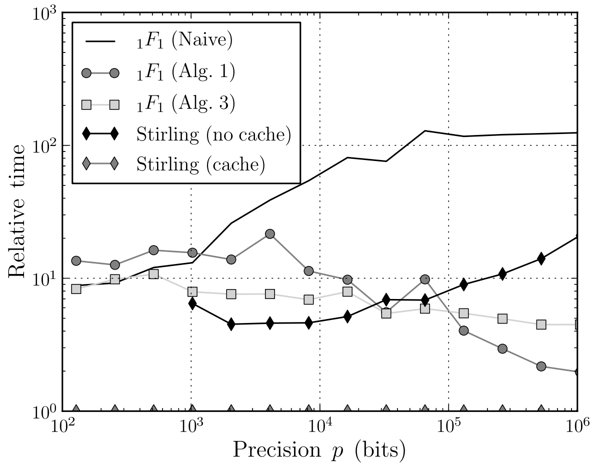

Figure 2: Timings of gamma function algorithms, normalized against the Stirling series with Bernoulli numbers cached.

Both fast multipoint evaluation and rectangular splitting speed up the hypergeometric series compared to naive summation. Using either algorithm, the hypergeometric series is competitive with the Stirling series for a single evaluation at precisions above roughly 10,000 decimal digits.

Algorithm 1 performs better than on the rising factorial benchmark, and is faster than Algorithm 3 above 10^5 bits. A possible explanation for this difference is that the number of terms used in the hypergeometric series roughly is $n \approx 2p$ where p is the precision in bits, compared to $n \approx p/4$ for the rising factorial, and rectangular splitting favors higher precision and fewer terms.

The speed of Algorithm 3 is remarkably close to that of Algorithm 1 even for p as large as 10^6. Despite being asymptotically slower, the simplicity of rectangular splitting combined with its lower memory consumption and better numerical stability (in our implementation, Algorithm 3 only loses a few significant digits, while Algorithm 1 loses a few percent of the number of significant digits) makes it an attractive option for extremely high-precision evaluation of the gamma function.

Once the Bernoulli numbers have been cached after the first evaluation, the Stirling series still has a clear advantage up to precisions exceeding 10^6 bits. We may remark that our implementation of the Stirling series has been optimized for multiple evaluations: by choosing larger rising factorials and generating the Bernoulli numbers dynamically without storing them, both the speed and memory consumption for a single evaluation could be improved.

6. DISCUSSION

We have shown that rectangular splitting can be profitably applied to evaluation of a general class of linearly recurrent sequences. When used for numerical evaluation of special functions, our benchmark results indicate that rectangular splitting can be faster than either naive evaluation or fast multipoint evaluation over a wide precision range (between approximately 10^3 and 10^6 bits).

Two natural questions are whether this approach can be generalized further (to more general classes of sequences), and whether it can be optimized further (perhaps for more specific classes of sequences).

7. ACKNOWLEDGEMENTS

The author thanks Manuel Kauers for providing useful feedback on draft versions of this paper.

8. REFERENCES

[1] A. V. Aho, K. Steiglitz, and J. D. Ullman. Evaluating polynomials at fixed sets of points. *SIAM Journal on Computing*, 4(4):533–539, 1975.

[2] D. J. Bernstein. Fast multiplication and its applications. *Algorithmic Number Theory*, 44:325–384, 2008.

[3] P. B. Borwein. Reduced complexity evaluation of hypergeometric functions. *Journal of Approximation Theory*, 50(3), July 1987.

[4] A. Bostan, P. Gaudry, and É. Schost. Linear recurrences with polynomial coefficients and application to integer factorization and Cartier-Manin operator. *SIAM Journal on Computing*, 36(6):1777–1806, 2007.

[5] R. P. Brent and P. Zimmermann. *Modern Computer Arithmetic*. Cambridge University Press, 2011.

[6] D. G. Cantor and E. Kaltofen. On fast multiplication of polynomials over arbitrary algebras. *Acta Informatica*, 28(7):693–701, 1991.

[7] D. V. Chudnovsky and G. V. Chudnovsky. Approximations and complex multiplication according to Ramanujan. In *Ramanujan Revisited*, pages 375–472. Academic Press, 1988.

[8] W. B. Hart. Fast Library for Number Theory: An Introduction. In *Proceedings of the Third international congress conference on Mathematical software*, ICMS'10, pages 88–91, Berlin, Heidelberg, 2010. Springer-Verlag. `http://flintlib.org`.

[9] S. Köhler and M. Ziegler. On the stability of fast polynomial arithmetic. In *Proceedings of the 8th Conference on Real Numbers and Computers*, Santiago de Compostela, Spain, 2008.

[10] M. Mezzarobba. NumGfun: a package for numerical and analytic computation with D-finite functions. In *Proceedings of ISSAC'10*, pages 139–146, 2010.

[11] M. S. Paterson and L. J. Stockmeyer. On the number of nonscalar multiplications necessary to evaluate polynomials. *SIAM Journal on Computing*, 2(1), March 1973.

[12] D. M. Smith. Efficient multiple-precision evaluation of elementary functions. *Mathematics of Computation*, 52:131–134, 1989.

[13] D. M. Smith. Algorithm: Fortran 90 software for floating-point multiple precision arithmetic, gamma and related functions. *Transactions on Mathematical Software*, 27:377–387, 2001.

[14] J. van der Hoeven. Fast evaluation of holonomic functions. *TCS*, 210:199–215, 1999.

[15] J. van der Hoeven. Making fast multiplication of polynomials numerically stable. Technical Report 2008-02, Université Paris-Sud, Orsay, France, 2008.

[16] J. von zur Gathen and J. Gerhard. Fast algorithms for Taylor shifts and certain difference equations. In *Proceedings of ISSAC'97*, pages 40–47, 1997.

[17] J. von zur Gathen and J. Gerhard. *Modern Computer Algebra*. Cambridge University Press, 2nd edition, 2003.

[18] M. Ziegler. Fast (multi-)evaluation of linearly recurrent sequences: Improvements and applications. 2005. `http://arxiv.org/abs/cs/0511033`.