SIAM J. SCI. COMPUT.
Vol. 40, No. 6, pp. C726–C747

FAST AND RIGOROUS ARBITRARY-PRECISION COMPUTATION OF GAUSS–LEGENDRE QUADRATURE NODES AND WEIGHTS[*]

FREDRIK JOHANSSON[†] AND MARC MEZZAROBBA[‡]

Abstract. We describe a strategy for rigorous arbitrary-precision evaluation of Legendre polynomials on the unit interval and its application in the generation of Gauss–Legendre quadrature rules. Our focus is on making the evaluation practical for a wide range of realistic parameters, corresponding to the requirements of numerical integration to an accuracy of about 100 to 100 000 bits. Our algorithm combines the summation by rectangular splitting of several types of expansions in terms of hypergeometric series with a fixed-point implementation of Bonnet's three-term recurrence relation. We then compute rigorous enclosures of the Gauss–Legendre nodes and weights using the interval Newton method. We provide rigorous error bounds for all steps of the algorithm. The approach is validated by an implementation in the Arb library, which achieves order-of-magnitude speedups over previous code for computing Gauss–Legendre rules suitable for precisions in the thousands of bits.

Key words. Legendre polynomials, Gauss–Legendre quadrature, arbitrary-precision arithmetic, interval arithmetic

AMS subject classifications. 65Y99, 65G99, 33C45

DOI. 10.1137/18M1170133

1. Introduction.

The Legendre polynomials $P_n(x)$ are the sequence of orthogonal polynomials with respect to the unit weight on the interval $(-1, 1)$, normalized so that $P_n(1) = 1$. Like other classical orthogonal polynomials, Legendre polynomials satisfy a three-term recurrence, in this case the relation

$$(n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) = 0, \quad n \geq 1, \tag{1}$$

also known as Bonnet's formula, and a second-order differential equation, here

$$(1 - x^2)P_n''(x) - 2xP_n'(x) + n(n+1)P_n(x) = 0. \tag{2}$$

The definition implies that P_n has n roots all located in $(-1, 1)$. Perhaps the most important application of Legendre polynomials is the Gauss–Legendre quadrature rule

$$\int_{-1}^{1} f(x)\mathrm{d}x \approx \sum_{i=0}^{n-1} w_i f(x_i), \qquad w_i = \frac{2}{(1 - x_i^2)[P_n'(x_i)]^2}, \tag{3}$$

where the *nodes* x_i are the roots of P_n. The quantity w_i is called the *weight* associated with the node x_i.

For some applications in computer algebra, number theory, mathematical physics, and experimental mathematics, it is necessary to compute integrals to an accuracy of

[*]Submitted to the journal's Software and High-Performance Computing section February 9, 2018; accepted for publication (in revised form) September 26, 2018; published electronically November 27, 2018.
http://www.siam.org/journals/sisc/40-6/M117013.html
Funding: The work of the second author was supported in part by ANR grant ANR-14-CE25-0018-01 (FastRelax).
[†]INRIA – LFANT, CNRS – IMB – UMR 5251, Université de Bordeaux, 33400 Talence, France (fredrik.johansson@gmail.com).
[‡]Sorbonne Université, CNRS, Laboratoire d'Informatique de Paris 6, LIP6, F-75005 Paris, France (marc@mezzarobba.net).

hundreds of digits, and occasionally even tens of thousands of digits [2, 3, 9, 23]. The Gauss–Legendre formula (3) achieves an accuracy of p bits using $n = \mathcal{O}(p)$ evaluation points if f is analytic on a neighborhood of $(-1, 1)$, and if the neighborhood is large (that is, if the path of integration is well isolated from any singularities of f), then the constants hidden in the $\mathcal{O}$ notation are close to the best achievable by any quadrature rule [22]. This quality is related to the fact that (3) maximizes the order of accuracy among n-point quadrature rules for integrating polynomials. Indeed, it is exact when f is any polynomial of degree up to $2n - 1$. As a result, the accuracy is also excellent for analytic integrands that are well approximated by polynomials.[1]

In general, the error in (3) can be bounded in terms of $\sup_{x\in(-1,1)} |f^{(2n)}(x)|$, or, if f is analytic on an elliptical domain D with foci at ± 1, in terms of $\sup_{z\in D} |f(z)|$ and the semiaxes of the ellipse. Even when the conditions are not ideal for using (3) directly, rapid convergence is often possible by combining (3) with adaptive subdivision of the integration path [28]. We give some elements of comparison between Gauss–Legendre quadrature and alternative methods, such as Clenshaw–Curtis quadrature, in section 8.

The Gauss–Legendre scheme has the drawback that the quadrature nodes and weights are somewhat inconvenient to compute. Indeed, P_n becomes highly oscillatory for large n and hence presents difficulties for naive root-finding and polynomial evaluation methods. The classical Golub–Welsch algorithm avoids accuracy problems by formulating the task of computing the nodes as finding the eigenvalues of a tridiagonal matrix [13], but this approach is still too slow to be practical for large n.

In the last decade, several authors have contributed to the development of asymptotic methods that permit computing any individual node and weight for arbitrarily large n in $\mathcal{O}(1)$ time, culminating in the 2014 work by Bogaert [6, 15, 5]. For a review of this progress, see Townsend [32]. Of course, the "$\mathcal{O}(1)$" bound assumes that a fixed level of precision is used. In the prevailing literature this generally means 53-bit IEEE 754 floating-point arithmetic. In addition, the available $\mathcal{O}(1)$ implementations rely in part on heuristic error estimates without rigorously proved bounds.

The literature on arbitrary precision or rigorous evaluation is comparatively limited. Petras [27] gave explicit bounds for the error $|x_k^{(i)} - x_k|$ when the roots x_k of the Legendre polynomial P_n are approximated using Newton iteration

$$x_k^{(i+1)} = x_k^{(i)} - \frac{P_n(x_k^{(i)})}{P_n'(x_k^{(i)})} \tag{4}$$

provided that the initial values $x_k^{(0)}$ are computed by a certain asymptotic formula. However, Petras did not address the numerical evaluation of $P_n(x)$. Fousse [12] discussed the rigorous implementation of Gauss–Legendre quadrature using generic polynomial root isolation methods together with interval Newton iteration for root refinement, but did not study fast methods for large n. Code for high-precision Gauss–Legendre quadrature rules can also be found in packages such as Pari/GP [31] and ARPREC [4], but without error analysis and without special techniques for large n.

In the present article, we are interested in computing Gauss–Legendre nodes and weights to precisions p significantly larger than machine precision—typically in the hundreds to thousands of bits—with rigorous error bounds. As mentioned earlier,

[1] However, statements about the near-optimality of Gauss–Legendre quadrature must not be overinterpreted. Indeed, the rate of convergence of Gauss–Legendre quadrature is *not* optimal asymptotically when $n \to \infty$ for analytic f on a fixed neighborhood, being improvable by a small factor [34].

certain applications require enclosing the values of integrals to accuracies of this order, and it is often reasonable to use quadrature rules of degree n which grows roughly linearly with p for this purpose. For example, Johansson and Blagouchine [20] study the computation of Stieltjes constants to precisions of hundreds of digits using complex integration, building on, among other things, the work described in the present paper.

If we assume that the precision p varies, basic arithmetic operations are no longer constant-time. It is well known that addition, multiplication, and division of p-bit numbers (of bounded exponent) can be performed in $\widetilde{\mathcal{O}}(p)$ operations [8], where the notation $\widetilde{\mathcal{O}}(\cdot)$ means that we neglect logarithmic factors. It is then clear that any node and weight can be computed to p-bit accuracy in $\widetilde{\mathcal{O}}(np)$ time by performing $\mathcal{O}(\log p)$ Newton iterations (4) from an appropriate initial value. As a consequence, the full set of nodes and weights for the degree-n quadrature rule can be computed in $\widetilde{\mathcal{O}}(n^2 p)$ time. For numerical integration of analytic functions where we typically have $p \approx n$, a better (indeed, optimal) estimate than the classical $\widetilde{\mathcal{O}}(n^3)$ bound is possible.

THEOREM 1. *If $p \sim \alpha n$ for some fixed α, then the Gauss–Legendre nodes and weights of degree n can be computed to p-bit accuracy in $\widetilde{\mathcal{O}}(n^2)$ (equivalently, $\widetilde{\mathcal{O}}(p^2)$) bit operations.*

Proof sketch. Using the formulae in [27], we can compute good initial values for Newton iteration in $\widetilde{\mathcal{O}}(n)$ bit operations. The Newton iteration can be performed for all roots simultaneously using fast multipoint evaluation, which costs $\widetilde{\mathcal{O}}(np)$ bit operations. Fast multipoint evaluation is numerically unstable and generically loses $\mathcal{O}(n)$ bits of accuracy, but we can compensate for this loss by using $\mathcal{O}(n)$ guard bits [21]. Since $p \sim \alpha n$ by assumption, this does not change the complexity bound. ∎

Completing the details of the proof is a technical exercise. Despite being elegant in theory, the algorithm behind Theorem 1 has a high overhead in practice, in large part due to the need to work with greatly increased precision. Working with expanded polynomials and processing all roots simultaneously also results in high memory usage and makes parallelization difficult. As discussed in subsection 5.2 below, an approach based on the "bit-burst" evaluation method for hypergeometric series leads to a similar complexity bound and may prove more practical for extremely large p, but likely not for $p \leq 10^6$. We can achieve a slightly worse but still subcubic complexity of $\widetilde{\mathcal{O}}(n^{5/2})$ by employing fast multipoint evaluation in a completely different way to compute P_n values in isolation [16], but unfortunately that algorithm also has high overhead.

In this work, we develop rigorous and more practical alternatives to the asymptotically fast algorithm outlined above. Our main contribution is to give a complete evaluation strategy for Legendre polynomials on $[-1, 1]$ in ball arithmetic [35, 19]. Computing the Gauss–Legendre nodes, then, is a relatively simple application of the results in [27] together with the interval Newton method [24]. For generating Gauss–Legendre quadrature rules with $n \sim \alpha p$, our algorithm has an asymptotic bit complexity of $\widetilde{\mathcal{O}}(n^3)$ like classical methods, but much lower overhead. For parameters $p, n \leq 10^5$ which are most relevant to applications, the observed running time is effectively subcubic. Our algorithm outperforms that of Theorem 1 for practically any combination of n and p in that range. Furthermore, if $p = \mathcal{O}(1)$, the complexity reduces to $\widetilde{\mathcal{O}}(n)$ as in the machine-precision implementations by Bogaert [5] and others.

The remainder of this paper is organized as follows. Section 2 gives an overview of our algorithm for evaluating Legendre polynomials. This is a hybrid algorithm that switches between different methods, detailed in the following sections, depending on the values of n, p, and x. In section 3, we prove practical error bounds for the three-term recurrence (1), which can be efficiently implemented in fixed-point arithmetic. This method is ideal for n and p up to a few hundred. For larger n or p, we use a fast method for evaluation of hypergeometric series. Section 4 discusses the hypergeometric series expansions that are preferable for different inputs and precision (including a well-known asymptotic expansion for large n), and section 5 their efficient evaluation. In section 6, we propose a strategy to select the best formula for any combination of n, p, x. Section 7 presents benchmark results that compare the performance of our algorithm to some previous implementations as well as the asymptotically fast algorithm in Theorem 1. Finally, section 8 reviews the viability of Gauss–Legendre quadrature compared to other methods for extremely high precision integration.

Our code for evaluating Legendre polynomials and computing Gauss–Legendre nodes and weights is freely available as part of the Arb library [19].[2]

2. General strategy. We work in the framework of midpoint-radius interval arithmetic, also called ball arithmetic [35, 19]. In general, given an integer n and a ball $x = [m \pm r] = [m - r, m + r]$, we want to evaluate $P_n(x)$ at x, yielding an enclosure $y = [m' \pm r']$ such that $P_n(\xi) \in y$ holds for all $\xi \in x$.

We restrict our attention to real $x \in [-1, 1]$, which is the most interesting part of the domain for applications. Since $P_n(-x) = (-1)^n P_n(x)$, we can further restrict to $0 \le x \le 1$. Bogaert [5] suggests working with $P_n(\cos(\theta))$ instead of $P_n(x)$ directly to improve numerical stability for x close to ± 1. This is not necessary in arbitrary-precision arithmetic since a slight precision increase (of the order of $\mathcal{O}(\log n)$ bits, since the distance between two successive roots of P_n close to ± 1 is about $1/n^2$) works as well.

We note that, for rigorous evaluation of $P_n(z)$ with complex z as well as Legendre functions of noninteger order n, generic methods for the hypergeometric ${}_2F_1$ function are applicable if n is not extremely large; see [18]. Real $|x| > 1$ can also be easily handled using naive methods.

In view of the use of Newton's method to compute the roots, we also need to evaluate the derivative $P_n'(x)$, typically at the same time as $P_n(x)$ itself. A simple option is to deduce $P_n'(x)$ from $P_n(x)$ and $P_{n-1}(x)$ using

$$(x^2 - 1)P_n'(x) = n\big(xP_n(x) - P_{n-1}(x)\big). \tag{5}$$

When x is close to ± 1, though, this formula involves a cancellation of about $|\log_2(1-x)|$ bits in the subtraction, followed by a division by $x^2 - 1$. Therefore, a direct evaluation of $P_n'(x)$ may be preferable to reduce the working precision. We use either of these strategies depending on the values of n, p, and x.

Our evaluation algorithms rely on ball arithmetic internally to propagate the error bounds up to the final result. Therefore, to ensure that the enclosure output by our evaluation algorithm contains the image of the input, it is enough to have bounds on the truncation errors associated to each of the approximate expressions of P_n that we use. The corresponding bounds are stated in (19), (26), and (29).

To limit overestimation and computational overhead, we deviate from the direct use of ball arithmetic on two occasions. First, Algorithm 1 does not keep track of round-off errors internally: instead, we prove an a priori bound on the accumulated error (Corollary 6) and add it to the radius of the output ball after calling that algorithm. Second, since some methods would produce unsatisfactorily large enclosures when executed on input balls $x = [m \pm r]$ of radius $r > 0$, we evaluate $P_n(m)$ (with

[2]http://arblib.org/

higher internal precision if necessary) and $P_n'(x)$ and use a first-order bound

$$\max_{\xi \in x} |P_n(\xi) - P_n(m)| \le r \max_{\xi \in x} |P_n'(\xi)|$$

to separately bound the propagated error. Similarly, we use a bound for P_n'' to compute a reasonably tight enclosure for $P_n'([m \pm r])$. Suitable bounds are given in Proposition 3. Denote by $[z^n]f(z)$ the coefficient of index n in a power series $f(z)$, and write $f(z) \ll_z \hat{f}(z)$ for two power series $f, \hat{f}$ if $\hat{f}$ has nonnegative coefficients and $|[z^n]f(z)| \le [z^n]\hat{f}(z)$ for all n.

Lemma 2. *If f, g, $\hat{f}$, $\hat{g}$ are such that $f(z) \ll_z \hat{f}(z)$ and $g(z) \ll_z \hat{g}(z)$, then $\int_0^z f \ll_z \int_0^z \hat{f}$ and $f(z)g(z) \ll_z \hat{f}(z)\hat{g}(z)$.*

Proposition 3. *The following bounds hold for $-1 \le x \le 1$:*

$$|P_n'(x)| \le \min\left(\frac{2^{3/2}}{\sqrt{\pi}} \frac{\sqrt{n}}{(1-x^2)^{3/4}}, \frac{n(n+1)}{2}\right), \tag{6}$$

$$|P_n''(x)| \le \min\left(\frac{2^{5/2}}{\sqrt{\pi}} \frac{n^{3/2}}{(1-x^2)^{5/4}}, \frac{(n-1)n(n+1)(n+2)}{8}\right). \tag{7}$$

Proof. It is classical that Legendre polynomials are given by the generating series

$$F(x,z) = \sum_{n=0}^{\infty} P_n(x) z^n = \frac{1}{\sqrt{1-2xz+z^2}}. \tag{8}$$

Differentiation with respect to x yields

$$\sum_{n=0}^{\infty} P_n'(x) z^n = \frac{zF(x,z)}{1-2xz+z^2}, \qquad \sum_{n=0}^{\infty} P_n''(x) z^n = \frac{3z^2 F(x,z)}{(1-2xz+z^2)^2}.$$

Set $\theta = \arccos x$, so that the roots of $1 - 2xz + z^2$ are $e^{\pm i\theta}$. Then, in the notation of Lemma 2, we have the bound

$$\frac{1}{1-2xz+z^2} = \frac{1}{2i\sin\theta}\left(\frac{1}{z-e^{i\theta}} - \frac{1}{z-e^{-i\theta}}\right) = \sum_{n=0}^{\infty} \frac{\sin\big((n+1)\theta\big)}{\sin(\theta)} z^n \ll_z \frac{\sin(\theta)^{-1}}{1-z}.$$

In addition, Bernstein's inequality for the Legendre polynomials [1] (see also [10]) combined with the logarithmic convexity of the Gamma function yields

$$|P_n(x)| \le \frac{\sqrt{2}}{\sqrt{\pi}} \frac{1}{\sqrt{\sin\theta}} \frac{1}{\sqrt{n+1/2}} \le \frac{\sqrt{2}}{\sqrt{\pi}} \frac{1}{\sqrt{\sin\theta}} \frac{\Gamma(n+1/2)}{\Gamma(n+1)},$$

and hence

$$F(x,z) \ll_z \frac{\sqrt{2}}{\sqrt{\sin\theta}} \sum_{n=0}^{\infty} \frac{1}{\sqrt{\pi}} \frac{\Gamma(n+1/2)}{\Gamma(n+1)} z^n = \frac{\sqrt{2}}{\sqrt{\sin\theta}} \frac{1}{\sqrt{1-z}}.$$

By Lemma 2, these bounds combine into

$$\frac{\mathrm{d}F}{\mathrm{d}x} \ll_z \frac{\sqrt{2}}{\sin(\theta)^{3/2}} \frac{1}{(1-z)^{3/2}}, \qquad \frac{\mathrm{d}^2F}{\mathrm{d}x^2} \ll_z \frac{\sqrt{2}}{\sin(\theta)^{5/2}} \frac{3z^2}{(1-z)^{5/2}}.$$

Since $[z^n](1-z)^{-k-1/2} = \Gamma(n+1/2)/(\Gamma(k+1/2)\Gamma(n-k+1))$ and using again the logarithmic convexity of Γ, we conclude that

$$|P_n'(x)| \leq \frac{\sqrt{2}}{\sin(\theta)^{3/2}} \frac{2}{\sqrt{\pi}} \frac{\Gamma(n+1/2)}{\Gamma(n)} \leq \frac{2^{3/2}}{\sqrt{\pi}} \frac{\sqrt{n}}{\sin(\theta)^{3/2}}, \qquad |P_n''(x)| \leq \frac{2^{5/2}}{\sqrt{\pi}} \frac{n^{3/2}}{(1-x^2)^{5/4}}.$$

The result follows since all derivatives of Legendre polynomials reach their maximum at $x=1$ (or by using the bounds $(z-e^{\pm i\theta})^{-1}, F(x,z) \ll_z (1-z)^{-1}$ and Lemma 2). □

Remark 4. By the same reasoning, the inequality

$$|P_n^{(k)}(x)| \leq \frac{2^{k+1/2}}{\sqrt{\pi}} \frac{n^{k-1/2}}{(1-x^2)^{(2n+1)/4}}$$

actually holds for all k. Unfortunately, it seems to overestimate the envelope of $|P_n^{(k)}|$ by a factor about 2^k in the region where it is smaller than $P_n^{(k)}(1)$.

Based on these reductions, we assume from now on that x is a floating-point number with $0 \leq x \leq 1$. Our main algorithm for evaluating P_n at x combines the following methods:

- the iterative computation of $P_n(x)$ via the three-term recurrence (1);
- the asymptotic expansion (17) of $P_n(x)$ as $n \to \infty$;
- the usual expanded expression (23), (24) of P_n in the monomial basis;
- the analogous terminating expansion (27) at 1.

All three expansions can be written as hypergeometric series, i.e., sums of the form $\sum_k c_k \xi^k$, where c_k/c_{k-1} is a rational function of k.

The constraints and heuristics used to select from among these methods are described in detail below. Roughly speaking, the three-term recurrence is used for small index n and precision p, when x is not too close to 1; the asymptotic series when n is large enough, again with x not too close to 1; the expansion at 0 for large p unless x is close to 1; and finally the expansion at 1 in the remaining cases when x is close to 1.

For an evaluation at p-bit precision, we choose parameters such as truncation orders and internal working precision to target an absolute error of $2^{-p-p'}$ for some $p' = \mathcal{O}(\log n)$, corresponding to a relative error of about 2^{-p} measured with respect to monotone envelopes for $P_n(x)$ and $P_n'(x)$ as in [6]. The relative error of a computed ball for $P_n(x)$ where x is near a zero x_k can be arbitrarily large, but the relative error of $P_n'(x)$ near x_k will be small, which is sufficient for the Newton iteration method. Since the output consists of a ball, we also have the option of catching a result with large relative error and repeating the evaluation with a higher precision as needed.

3. Basecase recurrence.

For small n, a straightforward way to compute $P_n(x)$ is to apply the three-term recurrence (1), starting from $P_0(x)=1$ and $P_1(x)=x$. Computing $P_n(x)$ by this method takes about $(\mathcal{M}(t)+\mathcal{O}(t))\,n$ bit operations, where t is the working precision and $\mathcal{M}(t)$ denotes the cost of t-bit multiplication. It is thus attractive for small n and t, especially when both $P_n(x)$ and $P_n'(x)$ are needed, since we can get $P_{n-1}(x)$ at no additional cost.

Fix $x \in [-1,1]$, and let $p_n = P_n(x)$. Bonnet's formula (1) gives

$$p_{n+1} = \frac{1}{n+1}\big((2n+1)xp_n - np_{n-1}\big), \qquad n \geq 0. \tag{9}$$

In a direct implementation of this recurrence in ball arithmetic, the width of the enclosures would roughly double at every iteration, requiring us to increase the internal working precision by $\mathcal{O}(n)$ bits. We avoid this issue by performing an a priori round-off error analysis (to be presented now) of the evaluation that yields a less pessimistic bound on the accumulated error. Additionally, the static error bound allows us to implement the recurrence in fixed-point arithmetic, avoiding the overhead of floating-point and interval operations.

Suppose $x = \hat{x}\, 2^{-t}$ with $\hat{x} \in \mathbb{Z}$ is a given fixed-point number. Let $\lceil u \rfloor$ denote the integer truncation of a real number u (note that this is not the same thing as rounding to the nearest integer; however, any rounding function would do). The integer sequence $(\hat{p}_n)$ defined by

$$\hat{p}_0 = 2^t, \qquad \hat{p}_1 = \hat{x}, \qquad \hat{p}_{n+1} = \left\lceil \frac{1}{n+1}\big((2n+1)\lceil \hat{x}\hat{p}_n 2^{-t} \rfloor - n\hat{p}_{n-1}\big) \right\rfloor \tag{10}$$

is easy to compute using only integer arithmetic, and $\hat{p}_n\, 2^{-t}$ is an approximation of p_n. Algorithm 1 provides a complete C implementation using GMP [14]. As a small optimization, we delay the division by $n+1$ until we have accumulated a denominator of the size of a machine word.

Algorithm 1. Evaluation of Legendre polynomials in GMP fixed-point arithmetic.

Input: An integer x and $t \geq 0$ such that $|2^{-t}x| \leq 1$, and $n \geq 1$
Output: p, q such that $|2^{-t}p - P_{n-1}(2^{-t}x)|, |2^{-t}q - P_n(2^{-t}x)| \leq (0.75\,(n+1)(n+2)+1)\, 2^{-t}$

```
void legendre(mpz_t p, mpz_t q, int n, const mpz_t x, int t) {
    mpz_t tmp; int k; mpz_init(tmp);          ▷ Comments use the notation of
    mp_limb_t denlo, den = 1;                 ▷ the proof of Corollary 6
    mpz_set_ui(p, 1); mpz_mul_2exp(p, p, t);                  ▷ $\mathtt{p}_0 = 2^t$
    mpz_set(q, x);                                            ▷ $\mathtt{q}_0 = \hat{x}$
    for (k = 1; k < n; k++) {
        mpz_mul(tmp, q, x); mpz_tdiv_q_2exp(tmp, tmp, t);     ▷ $\lceil \hat{x}\, \mathtt{q}_{k-1}\, 2^{-t} \rfloor$
        mpz_mul_si(p, p, -k*k)
        mpz_addmul_ui(p, tmp, 2*k+1);             ▷ $-k^2 \mathtt{p}_{k-1} + (2k+1)\,\mathtt{tmp}$
        mpz_swap(p, q);
        if (mpn_mul_1(&denlo, &den, 1, k+1)) {    ▷ If multiplication overflows
            mpz_tdiv_q_ui(p, p, den);                         ▷ $\lceil \mathtt{p}/\mathtt{d}_{k-1} \rfloor$
            mpz_tdiv_q_ui(q, q, den);
            den = k+1;                                        ▷ $\mathtt{d}_k = k+1$
        } else den = denlo;                       ▷ $\mathtt{d}_k = (k+1)\,\mathtt{d}_{k-1}$
    }
    mpz_tdiv_q_ui(p, p, den/n); mpz_tdiv_q_ui(q, q, den);
    mpz_clear(tmp);
}
```

To bound the difference $|\hat{p}_n\, 2^{-t} - p_n|$, we analyze the effect on the result of a small perturbation in each iteration of (9). The bound is based on a classical linearity argument (compare with, e.g., [36]) combined with majorant series techniques.

Proposition 5. *Suppose that a real sequence $(\tilde{p}_n)_{n \geq -1}$ satisfies $\tilde{p}_0 = 1$ and*

$$\tilde{p}_{n+1} = \frac{1}{n+1}\big((2n+1)x\tilde{p}_n - n\tilde{p}_{n-1}\big) + \varepsilon_n, \qquad n \geq 0, \tag{11}$$

for arbitrary real numbers ε_n with $|\varepsilon_n| \leq \bar{\varepsilon}$ for all n. Then we have

$$|\tilde{p}_n - P_n(x)| \leq \frac{(n+1)(n+2)}{4}\bar{\varepsilon}$$

for all $n \geq 0$.

Proof. Let $\delta_n = \tilde{p}_n - p_n$ and $\eta_n = (n+1)\varepsilon_n$. Subtracting (9) from (11) gives

$$(n+1)\delta_{n+1} = (2n+1)x\delta_n - n\delta_{n-1} + \eta_n, \tag{12}$$

with $\delta_0 = 0$. Consider the formal generating series $\delta(z) = \sum_{n\geq 0} \delta_n z^n$ and $\eta(z) = \sum_{n\geq 0} \eta_n z^n$. Noting that (12) holds for all $n \in \mathbb{Z}$ if the sequences (δ_n) and (η_n) are extended by 0 for $n < 0$ and using the relations

$$\sum_{n=-\infty}^{\infty} f_{n-1} z^n = z \sum_{n=-\infty}^{\infty} f_n z^n, \qquad \sum_{n=-\infty}^{\infty} n f_n z^n = z \frac{\mathrm{d}}{\mathrm{d}z} \sum_{n=-\infty}^{\infty} f_n z^n,$$

we see that (12) translates into

$$(1 - 2xz + z^2) z \frac{\mathrm{d}}{\mathrm{d}z} \delta(z) = z(x - z)\delta(z) + z\eta(z).$$

The solution of this differential equation with $\delta(0) = 0$ reads (cf. (8))

$$\delta(z) = p(z) \int_0^z \eta(w)\, p(w)\, \mathrm{d}w, \qquad p(z) = \sum_{n=0}^{\infty} p_n z^n = F(x, z) = \frac{1}{\sqrt{1 - 2xz + z^2}}.$$

This is an exact expression of the "global" error δ in terms of the "local" errors ε_n. Since $|p_n| = |P_n(x)| \leq 1$ and $|\eta_n| \leq (n+1)\bar{\varepsilon}$, it follows by Lemma 2 that

$$|\delta_n| = \left| [z^n] \left(p(z) \int_0^z \eta(w)\, p(w)\, \mathrm{d}w \right) \right| \leq [z^n] \left(\frac{1}{1-z} \int_0^z \frac{\bar{\varepsilon}}{(1-w)^2} \frac{1}{1-w} \mathrm{d}w \right)$$

and therefore

$$|\delta_n| \leq [z^n] \left(\frac{1}{2} \frac{\bar{\varepsilon}}{(1-z)^3} \right) = \frac{(n+1)(n+2)}{4}\, \bar{\varepsilon}.$$

∎

Corollary 6. *Suppose that* $x = \hat{x}\, 2^{-t}$ *for some* $t \geq 0$ *and* $\hat{x} \in \mathbb{Z}$. *The sequence* $(\hat{p}_n)_{n\geq 0}$ *defined by* (10) *satisfies*

$$|\hat{p}_n 2^{-t} - p_n| \leq 0.75\, (n+1)(n+1)\, 2^{-t}, \qquad n \geq 0. \tag{13}$$

Furthermore, the quantities p, q *returned by Algorithm* 1 *are such that*

$$|p - 2^t P_{n-1}(x)|, |q - 2^t P_n(x)| \leq 0.75(n+1)(n+2) + 1. \tag{14}$$

Proof. We can write

$$\hat{p}_{n+1} = \frac{1}{n+1} \big((2n+1)(\hat{x}\, \hat{p}_n\, 2^{-t} + \alpha_n) - n\, \hat{p}_{n-1} \big) + \beta_n$$

for some α_n, β_n of absolute value at most one, and hence

$$\hat{p}_{n+1} = \frac{1}{n+1} \big((2n+1)\, \hat{x}\, \hat{p}_n\, 2^{-t} - n\, \hat{p}_{n-1} \big) + \varepsilon_n, \qquad \varepsilon_n = \frac{2n+1}{n+1} \alpha_n + \beta_n,$$

where $|\varepsilon_n| \leq 3$. Proposition 5 applied to $\tilde{p}_n = \hat{p}_n\, 2^{-t}$ then provides the bound (13).

Turning to Algorithm 1, let $\mathtt{p}_0$, $\mathtt{q}_0$, $\mathtt{d}_0$ denote the values of the variables `p`, `q`, `den` before the loop, and $\mathtt{p}_k$, $\mathtt{q}_k$, $\mathtt{d}_k$ their values at the end of iteration k. Consider the sequence $\tilde{p}_k = 2^{-t}\mathtt{q}_{k-1}/\mathtt{d}_{k-1}$, $k \geq 1$, extended by $\tilde{p}_0 = 1$ and an arbitrary (finite) $\tilde{p}_{-1}$. For all $k \geq 1$, depending whether the conditional branch is taken, we have one of the systems of equations

$$\mathtt{p}_k = \mathtt{q}_{k-1}, \qquad \mathtt{q}_k = (2k+1)\lceil \hat{x}\mathtt{q}_{k-1}2^{-t}\rfloor - k^2\mathtt{p}_{k-1}, \qquad \mathtt{d}_k = (k+1)\,\mathtt{d}_{k-1}, \tag{15}$$

$$\mathtt{p}_k = \left\lceil \frac{\mathtt{q}_{k-1}}{\mathtt{d}_{k-1}} \right\rfloor, \quad \mathtt{q}_k = \left\lceil \frac{(2k+1)\lceil \hat{x}\mathtt{q}_{k-1}2^{-t}\rfloor - k^2\mathtt{p}_{k-1}}{\mathtt{d}_{k-1}} \right\rfloor, \quad \mathtt{d}_k = k+1. \tag{16}$$

In both cases, we can write

$$\frac{\mathtt{p}_k}{\mathtt{d}_k} = \frac{\mathtt{q}_{k-1}}{(k+1)\mathtt{d}_{k-1}} + \frac{\alpha_k}{k+1}, \qquad \frac{\mathtt{q}_k}{\mathtt{d}_k} = \frac{(2k+1)(x\mathtt{q}_{k-1} + \beta_k) - k^2\mathtt{p}_{k-1}}{(k+1)\mathtt{d}_{k-1}} + \frac{\gamma_k}{k+1}$$

with $|\alpha_k|, |\beta_k|, |\gamma_k| \leq 1$. The first equation implies $2^{-t}k\,\mathtt{p}_{k-1}/\mathtt{d}_{k-1} = \tilde{p}_{k-1} + \alpha_{k-1}2^{-t}$ for $k \geq 2$. Since the latter equality also holds for $k = 1$ with $\alpha_0 = 0$, we can substitute it into the second equation, yielding

$$\tilde{p}_{k+1} = \frac{(2k+1)x\tilde{p}_k - k\tilde{p}_{k-1}}{k+1} + 2^{-t}\left(\frac{-k\alpha_{k-1}}{k+1} + \frac{(2k+1)\beta_k}{(k+1)\mathtt{d}_{k-1}} + \frac{\gamma_k}{k+1}\right).$$

This relation holds for $k \geq 1$, and we extend it to $k = 0$ by setting $\beta_0 = \gamma_0 = 0$. Thus, $\tilde{p}_k$ also satisfies (11) with $|\varepsilon_n| \leq 3 \cdot 2^{-t}$, and Proposition 5 applies. The final values of $\mathtt{q}$ and $\mathtt{p}$ are, respectively, $\lceil 2^t\tilde{p}_n\rfloor$ and $\lceil n\mathtt{p}_{n-1}/\mathtt{d}_{n-1}\rfloor = \lceil 2^t\tilde{p}_{n-1} + \alpha_{n-1}\rfloor$, whence the bound (14). ∎

We do not use asymptotically faster evaluation techniques for large n in combination with this recurrence since the series expansions to be presented next perform very well in this case.

4. Series expansions.

For large n or p, we employ series expansions of $P_n(x)$ with respect to either n or x rather than the algorithm from the previous section. The coefficients of the series are also computed by recurrence, but fewer than n terms will typically be required. Let us now review the various series expansions that we are using (an asymptotic expansion as $n \to \infty$, series expansions at $x = 0$ and $x = 1$) before discussing their efficient evaluation in the next section.

4.1. Asymptotic series.

For fixed $|x| < 1$, or equivalently $x = \cos(\theta)$ with $0 < \theta < \pi$, an asymptotic expansion for $P_n(x)$ as $n \to \infty$ can be given as [6, eq. (3.4)]

$$P_n(\cos(\theta)) = \left(\frac{2}{\pi\sin(\theta)}\right)^{1/2} \sum_{k=0}^{K-1} C_{n,k}\frac{\cos(\alpha_{n,k}(\theta))}{\sin^k(\theta)} + \xi_{n,K}(\theta) \tag{17}$$

(for arbitrary $K \geq 1$), where

$$C_{n,k} = \frac{[\Gamma(k+1/2)]^2\Gamma(n+1)}{\pi 2^k\Gamma(n+k+3/2)\Gamma(k+1)}, \tag{18}$$
$$\alpha_{n,k}(\theta) = (n+k+1/2)\theta - (k+1/2)\pi/2,$$

and the error term satisfies

$$|\xi_{n,K}(\theta)| < 2\left(\frac{2}{\pi\sin(\theta)}\right)^{1/2}\frac{C_{n,K}}{\sin^K(\theta)}. \tag{19}$$

The coefficients $C_{n,k}$ are a hypergeometric sequence with

$$\frac{C_{n,k}}{C_{n,k-1}} = \frac{(2k-1)^2}{4k(2n+2k+1)}, \quad C_{n,0} = \frac{1}{\sqrt{\pi}} \frac{4^n}{(n+\frac{1}{2})\binom{2n}{n}}. \tag{20}$$

To evaluate the error bound, we can use the following inequality deduced from (18):

$$C_{n,k} \le \frac{1}{\pi n^{1/2}} \frac{k!n!}{2^k(n+k)!} \le \frac{1}{\pi n^{1/2}} \frac{k!}{(2n)^k}, \quad n,k \ge 1. \tag{21}$$

The condition $|\xi_{n,K}(\theta)| \le 2^{-p}$ is satisfiable as soon as $\sin(\theta) \ge (p+3)\ln(2)/(2n)$, as we can see by choosing $K = \lfloor 2n\sin(\theta) \rfloor$ and using the inequality $k! \le k^{k+1/2}e^{1-k}$. Since the Gauss–Legendre nodes are distributed linearly in θ, the asymptotic expansion gives a candidate algorithm for all but about $(\log(2)/\pi)\, p/n$ of the nodes as $p/n \to 0$. It is in fact a convergent series when $2\sin(\theta) > 1$, which allows evaluating $P_n(x)$ to unbounded accuracy for fixed n when $\frac{1}{6}\pi < \theta < \frac{5}{6}\pi$. The particular form (17) must be used for this purpose; there is a slightly different version of the expansion which is asymptotic to $P_n(x)$ (for fixed K when $n \to \infty$) but paradoxically converges to $2P_n(x)$ (for fixed n when $K \to \infty$); see [25, section 10.3] and [26, section 18.15(iii)].

We can restate (17) as a hypergeometric series by working with complex numbers. Letting $\omega = 1 - (x/y)i$, with $x = \cos(\theta)$ and $y = \sin(\theta)$ as usual, we have

$$(1-i)(x+iy)^{n+1/2}\omega^k = \sqrt{2}\, e^{-i\frac{\pi}{4}}\, e^{i(n+1/2)\theta}\, e^{ik(\theta-\pi/2)}\, y^{-k} = \sqrt{2}\, \frac{\exp(i\,\alpha_{n,k}(\theta))}{\sin^k(\theta)},$$

and hence

$$P_n(x) = \frac{1}{\sqrt{\pi y}} \operatorname{Re}\left[(1-i)(x+yi)^{n+1/2} \sum_{k=0}^{K-1} C_{n,k}\omega^k\right] + \xi_{n,K}(\theta). \tag{22}$$

This eliminates the explicit trigonometric functions and permits using Algorithm 2 below to evaluate the series.

The evaluation of (22) in ball arithmetic is numerically stable, and we therefore only need to add a few guard bits to the working precision.

4.2. Expansion at zero. If $n = 2d$ is even, the expansion of $P_n(x)$ in the monomial basis reads

$$\begin{aligned} P_{2d}(x) &= (-1)^d \sum_{k=0}^{d} \frac{(-1)^k}{2^n} \binom{n}{d-k} \binom{n+2k}{n} x^{2k} \\ &= \frac{(-1)^d}{2^{2d}} \binom{2d}{d} \sum_{k=0}^{d} A_{-1}(d,k)(-x^2)^k, \end{aligned} \tag{23}$$

and if $n = 2d+1$ is odd, we have

$$\begin{aligned} P_{2d+1}(x) &= (-1)^d x \sum_{k=0}^{d} \frac{(-1)^k}{2^n} \binom{n}{d-k} \binom{n+2k+1}{n} x^{2k} \\ &= \frac{(-1)^d(d+1)}{2^{2d+1}} \binom{2d+2}{d+1}\, x \sum_{k=0}^{d} A_{+1}(d,k)(-x^2)^k, \end{aligned} \tag{24}$$

where the hypergeometric sequences $A_{\pm 1}$ can be defined by $A_{\pm 1}(d, 0) = 1$ and

$$\frac{A_\sigma(d,k)}{A_\sigma(d,k-1)} = \frac{(d-k+1)(2d+2k+\sigma)}{k(2k+\sigma)}, \quad \sigma \in \{-1,+1\}. \tag{25}$$

At very high precision, we evaluate the full polynomials, where (23) and (24) have the advantage of requiring only $n/2$ terms due to the odd-even form, whereas other expansions require n terms. At lower precision p, the high order terms will be smaller than 2^{-p} when $|x|$ is small, and we can truncate the series accordingly and add a bound for the omitted terms to the radius of the computed ball. When the series are truncated after the $k = K - 1$ term (for any $K < d + 1$), comparison with a geometric series shows that the error is bounded by the first omitted term times a simple factor.

Proposition 7. *For $\sigma \in \{-1, +1\}$, the error when truncating the bottom sum in* (23) *or* (24) *(with prefactors removed) after the $k = K - 1$ term satisfies*

$$\left| \sum_{k=K}^{d} A_\sigma(d,k)(-x^2)^k \right| \le \frac{A_\sigma(d,K)|x|^{2K}}{1-\alpha}, \quad \alpha = |x|^2 \frac{(d-K+1)(2d+2K+\sigma)}{K(2K+\sigma)} \tag{26}$$

provided that $\alpha < 1$.

For bounding $A_\sigma(d, K)$ in this expression, and for selecting an appropriate truncation point K, we use the binomial closed forms (23), (24) together with the remarks in subsection 5.3.

The alternating series (23) and (24) may suffer from significant cancellation, which requires the use of increased precision. We can estimate the magnitude by noting that no cancellation occurs if x is an imaginary number. Solving the majorizing recurrence $f_n = 2|z|f_{n-1} + f_{n-2}$ with $f_0 = 1$, $f_1 = |z|$ shows that

$$|P_n(z)| \le |P_n(i|z|)| \le \left(|z| + \sqrt{1+|z|^2}\right)^n.$$

Therefore, the possible cancellation assuming that $|P_n(x)| \approx 1$ is about

$$p_A = n \log_2\left(|x| + \sqrt{1+|x|^2}\right)$$

bits (which is at most $n \log_2(1 + \sqrt{2}) \approx 1.27\,n$), so using ball arithmetic with about $p + p_A$ bits of working precision for the series evaluation gives p-bit accuracy.

4.3. Expansion at one.

Expanding at $x = 1$ yields

$$P_n(x) = \sum_{k=0}^{n} c_{n,k} u^k, \quad c_{n,k} = \binom{n}{k}\binom{n+k}{k} \tag{27}$$

where $u = (x-1)/2$. The coefficients $c_{n,k}$ are hypergeometric with initial value $c_{n,0} = 1$ and term ratio

$$\frac{c_{n,k}}{c_{n,k-1}} = \frac{(n-k+1)(n+k)}{k^2}. \tag{28}$$

As in the previous section, we can truncate (27) and bound the error by comparison with a geometric series.

PROPOSITION 8. *The error when truncating* (27) *after the* $k = K-1$ *term satisfies*

$$\left|\sum_{k=K}^{n} c_{n,k} u^k\right| \le \frac{c_{n,K}|u|^K}{1-\alpha}, \qquad \alpha = |u|\frac{(n-K)(n+K+1)}{(K+1)^2} \tag{29}$$

provided that $\alpha < 1$.

For $u \ge 0$, (27) does not suffer from cancellation. For $u < 0$, we can estimate the amount of cancellation from the magnitude of $P_n(x')$ where $|u| = (x'-1)/2$. For not too large $x' \ge 1$, a very good approximation is

$$P_n(x') \le 2\sum_{k=0}^{\infty} \frac{n^{2k}}{k!^2}|u|^k = 2\, I_0(2n\sqrt{|u|}) \le 2\, e^{2n\sqrt{|u|}}.$$

We therefore need about $2n\sqrt{\max(0,-u)}/\ln(2)$ bits of increased precision.

We can compute P_n' from P_n and P_{n-1}, but since this involves a division by $1-x^2$, it is better to evaluate P_n' directly when x is close to 1. We have $P_n'(x) = \sum_{k=0}^{n-1} c'_{n,k} u^k$ where $c'_{n,k} = (k+1)c_{n,k+1}/2$ satisfies

$$c'_{n,0} = \frac{n(n+1)}{2}, \qquad \frac{c'_{n,k}}{c'_{n,k-1}} = \frac{(n-k)(n+k+1)}{k(k+1)}. \tag{30}$$

Since $c'_{n,k} \le n c_{n,k+1}$, the analogue

$$\left|\sum_{k=K}^{n} c'_{n,k} u^k\right| \le n\binom{n}{K+1}\binom{n+K+1}{K+1}|u|^K \frac{1}{1-\alpha} \tag{31}$$

of Proposition 8 holds with u and α as above.

5. Fast evaluation of series expansions.

All these series expansions are amenable to fast evaluation techniques specific to multiple-precision arithmetic. Using such fast summation algorithms is critical for achieving good performance at high precision. We now discuss the algorithm that we use for evaluating the series of the previous section.

5.1. Rectangular splitting.

We use rectangular splitting [29, 16] to evaluate hypergeometric series with rational parameters where the argument x is a high-precision number. This reduces evaluating a K-term series to $\mathcal{O}(K)$ cheap scalar operations (additions and multiplications or divisions by small integer coefficients) and about $2\sqrt{K}$ expensive nonscalar operations (general multiplications), whereas direct evaluation of the hypergeometric recurrence uses $\mathcal{O}(K)$ expensive operations.

Algorithm 2 presents our version of rectangular splitting for the present application. We implement the various series expansions by defining the functions $p(k), q(k)$ used in steps 6 and 17 according to formulae (20), (25), (28), (30).

This algorithm is a generalization of the method for evaluating Taylor series of elementary functions given in [17], which combines rectangular splitting with partially unrolling the recurrence to reduce the number of scalar divisions (which in practice are more costly than scalar multiplications). The terms are computed in the reverse direction to allow using Horner's rule for the outer multiplications.

Our code uses ball arithmetic for x and s so that no error analysis is needed, and we use a bignum type for c (so no overflow can occur regardless of u). For low

Algorithm 2. Evaluation of hypergeometric series using rectangular splitting.

```
Input: An arbitrary $x$, recurrence data $p, q \in \mathbb{Z}[k]$, integer $K \geq 0$, offset $\Omega \in \{0, 1\}$.
Output: $s = \sum_{k=\Omega}^{K-1} x^k \prod_{j=\Omega}^{k} p(j)/q(j)$
 1: $m \leftarrow \lfloor\sqrt{K}\rfloor$; precompute $[1, x, x^2, \ldots, x^m]$   ▷ Tuning parameter: any $m \geq 1$ can be used
 2: $s \leftarrow 0$;   $k \leftarrow K - 1$
 3: while $k \geq \Omega$ do
 4:     $u \leftarrow \min(4, k + 1 - \Omega)$   ▷ Tuning parameter: any $1 \leq u \leq k + 1 - \Omega$ can be used
 5:     $(a, b) \leftarrow (k - u + 1, k)$   ▷ Unrolled range
 6:     $c \leftarrow \prod_{j=a}^{b} p(j)$   ▷ Small integer coefficient
 7:     while $k \geq a$ do
 8:         $r \leftarrow k \bmod m$
 9:         if $k = b$ then
10:             $s \leftarrow c \cdot (s + x^r)$   ▷ Using precomputed power of $x$
11:         else
12:             $s \leftarrow s + c \cdot x^r$   ▷ Using precomputed power of $x$
13:         end if
14:         if $r = 0$ and $k \neq 0$ then
15:             $s \leftarrow s \cdot x^m$   ▷ Using precomputed power of $x$
16:         end if
17:         $c \leftarrow (c/p(k))q(k)$   ▷ Exact small integer division
18:         $k \leftarrow k - 1$
19:     end while
20:     $s \leftarrow s/c$
21: end while
22: return $s$
```

precision a faster implementation would be possible using fixed-point arithmetic with tight control of the word-level operations as was done for elementary functions in [17].

The algorithm contains two tuning parameters.[3] The splitting parameter m controls the number m of multiplications for powers versus the number K/m of multiplications for Horner's rule. The choice $m \approx \sqrt{K}$ is optimal, but when evaluating two series for the same x (in our case, to compute both $P_n(x)$ and $P'_n(x)$), the table of powers can be reused, and then $m \approx \sqrt{2K}$ minimizes the total cost.

The unrolling parameter u controls the number of coefficients to collect on a single denominator, reducing the number of divisions to N/u. Ideally, u should be chosen so that $\prod_{j=a}^{b} p(j)$ and $\prod_{j=a}^{b} q(j)$ fit in 1 or 2 machine words. The example value $u = 4$ is a reasonable default, but as an optimization, one might vary u for each iteration of the main loop to ensure that c always fits in a specific number of words.

The redundant parameter Ω is a small convenience in the pseudocode. Setting $\Omega = 1$ and adding the constant term separately avoids having to make a special case to prevent division by zero when $q(0) = 0$.

Due to the scalar operations, rectangular splitting ultimately requires $\mathcal{O}(K)$ arithmetic operations with $\widetilde{\mathcal{O}}(p)$ bit complexity each, just like straightforward evaluation of the recurrence, so it is not a genuine asymptotic improvement, but it is an improvement in practice and can give more than a factor-100 speedup at very high precision. It is possible to genuinely reduce the complexity of evaluating a hypergeometric sequence to $\mathcal{O}(\sqrt{K}\log(K))$ arithmetic operations using a baby-step giant-step method that employs fast multipoint evaluation, but in practice rectangular splitting performs

[3]Let us stress that the choice of these parameters only affects the performance of the algorithm—not its correctness. The same remark holds every other time we resort to heuristics in this article.

better until both K and p exceed 10^6 (see [16]).

5.2. A note on the bit-burst method. Another technique for fast evaluation of hypergeometric series, called binary splitting, would be useful when p is large and the argument x is a rational number with small numerator and denominator, but this case is not relevant for our application. Binary splitting also forms the basis of the bit-burst method [11, section 4], which permits evaluating any fixed hypergeometric series at any fixed point—without the restriction to simple rational numbers of plain binary splitting—to absolute precision p in only $\widetilde{\mathcal{O}}(p)$ bit operations. Yet, we do not use this method either in our implementation, due to its large overhead.

Indeed, computing $P_n(x)$ by the bit-burst method requires $\mathcal{O}(\log p)$ analytic continuation steps, each of which entails two evaluations of general solutions of the Legendre differential equation (2). These solutions are defined by unit initial values at some intermediate point x_0 and can be represented as a power series of radius of convergence $1 - |x|$ whose coefficients obey recurrences of order two. This is to be compared with a single series, given by a first-order recurrence and typically converging faster, for the expansions considered in subsections 4.1–4.3. Thus, the bit-burst method is unlikely to be competitive in the range of precision we are interested in, especially when the asymptotic series can be used.

Nevertheless, it can be shown that the solutions with unit initial values at x_0 of (2) occurring at intermediate analytic continuation steps have Taylor coefficients c_k at x_0 bounded by $(n^2/(1-|x_0|))^{\mathcal{O}(k)}$ uniformly in n and x_0. As a consequence, the asymptotic cost of computing any individual root of $P_n(x)$ by the bit-burst method and Newton iteration is $\widetilde{\mathcal{O}}(p)\ \log(n)^{\mathcal{O}(1)}$. For computing all the roots, this approach matches the $\widetilde{\mathcal{O}}(np)$ estimate of Theorem 1, while allowing for parallelization and requiring less memory. It may hence provide an alternative to multipoint evaluation worth investigating for precisions in the millions of bits.

5.3. Binomial coefficients. The prefactors of both the series expansion at $x = 0$ and the asymptotic series contain the central binomial coefficient $\binom{2n}{n}$. We need to compute this factor efficiently for any n and precision p. Since $\binom{2n}{n} \approx 4^n$, it is best to use an exact algorithm when $n < Cp$ for some small constant $C > 1/2$. We use the binomial function provided by GMP for $n < 6p + 200$ and otherwise use an asymptotic series for $\binom{2n}{n}$ with error bounds given in [7, Corollaries 1 and 2].

We also need to quickly estimate the magnitude of binomial coefficients for error bounds of series truncations. We have the binary entropy estimate

$$\log_2 \binom{n}{k} \le nG(k/n), \quad G(x) = -x\log_2(x) - (1-x)\log_2(1-x)$$

and the equivalent form

$$\binom{n}{k} \le \left(\frac{n}{n-k}\right)^{n-k} \left(\frac{n}{k}\right)^{k} = \frac{n^n}{k^k(n-k)^{n-k}}. \tag{32}$$

The function $G(x)$ can be evaluated cheaply with a precomputed lookup table. A coarse estimate is sufficient, since overestimating $\log_2 \binom{n}{k}$ by a few percent only adds a few percent to the running time.

6. Algorithm selection.

We first use a set of cutoffs found experimentally to decide whether to use the basecase recurrence or one of the series expansions. The recurrence is mainly faster for some combinations of $p < 1\,000$, $n < 400$ when computing $(P_n(x), P'_n(x))$ simultaneously and for some combinations of $p < 500$, $n < 100$ when computing $P_n(x)$ alone (in all cases subject to some boundaries $\varepsilon < x < 1 - \varepsilon$); the actual optimal regions are complicated due to differences in overhead between fixed-point integer arithmetic and ball arithmetic for the respective algorithm implementations. For the actual cutoffs used, we refer the reader to the source code.

To select between the series expansion at $x = 0$, the expansion at $x = 1$, and the asymptotic series, the following heuristic is used. For each algorithm A, we estimate the evaluation cost as $C_A = K_A(p + p_A)$, where K_A is the number of terms required by algorithm A ($K_A = \infty$ if A is the asymptotic series and it does not converge to the required accuracy), p is the precision goal, and p_A is the extra precision required by algorithm A due to internal cancellation. For the asymptotic series, we multiply the cost by an extra factor 2 as a penalty for using complex numbers. In the end, we select the algorithm with the lowest C_A.

We select K_A and estimate p_A heuristically using machine precision floating-point computations, working with logarithmic magnitudes to avoid underflow and overflow. For example, when A is the asymptotic series, we search for a small K_A such that $\log(K_A!/(2n \sin\theta)^{K_A}) \leq -(p+p_A)$ for some small p_A, in accordance with (21). During the actual evaluation of the series expansions, K_A and p_A are then given; we compute rigorous upper bounds for the truncation error via (19), (21), (26), (29), (31), (32) using floating-point arithmetic with directed rounding, while additional rounding errors are tracked by the ball arithmetic.

The assumption that the running time is a bilinear function of K_A and $p + p_A$ is not completely realistic, but this cost estimate nonetheless captures the correct asymptotics when

$$x \to 0, \quad x \to 1, \quad n \to \infty, \quad p \to \infty$$

separately and we hope will not be too inaccurate in the transition regions. This is verified empirically.

Figure 1 illustrates how the time to evaluate $(P_n(x), P'_n(x))$ varies with x when the automatic algorithm selection is used. Here, we have timed the case $n = 10\,000$ for two different p. For large n and $p \ll n$ (top plot), a sharp peak appears at the transition between the series expansion at $x = 1$ and the asymptotic expansion which is used for most x. This peak tends to become taller but narrower for larger n. We could presumably get rid of the peak by implementing another algorithm specifically for the transition region, but the area under the peak is so small compared to the median baseline that computing all the roots would not be sped up much. For p somewhat larger than n (bottom plot), we observe a smooth transition between the series at $x = 1$ near the left of the picture and the series at $x = 0$ used over the most of the range.

7. Benchmarks.

As already mentioned, our code for computing values and roots of Legendre polynomials is part of the Arb library, available from http://arblib.org/. The benchmark results given in this section were obtained using prerelease builds of version 2.13 of Arb. The implementation of the algorithms of this article is located in the files `arb_hypgeom/legendre_p_ui_*` of the Arb source tree. In subsection 7.1, we also use some ad hoc code available from this paper's public git repository[4] when comparing the main algorithm with variants absent from the Arb implementation. The source code for the experiments themselves can be found in the same git repository.

[4]https://github.com/fredrik-johansson/legendrepaper/

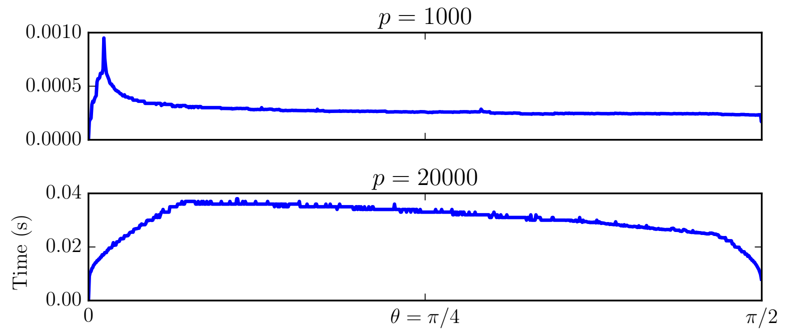

FIG. 1. *Time to evaluate $(P_n(x), P_n'(x))$ as the argument $x = \cos(\theta)$ varies ($x = 1$ at $\theta = 0$ and $x = 0$ at $\theta = \pi/2$), here with $n = 10\,000$, for precision p somewhat smaller than n (top plot) and somewhat larger (bottom plot). The variable θ is used for the horizontal scale in this picture to follow the distribution of the roots (which are clustered near $x = 1$) linearly.*

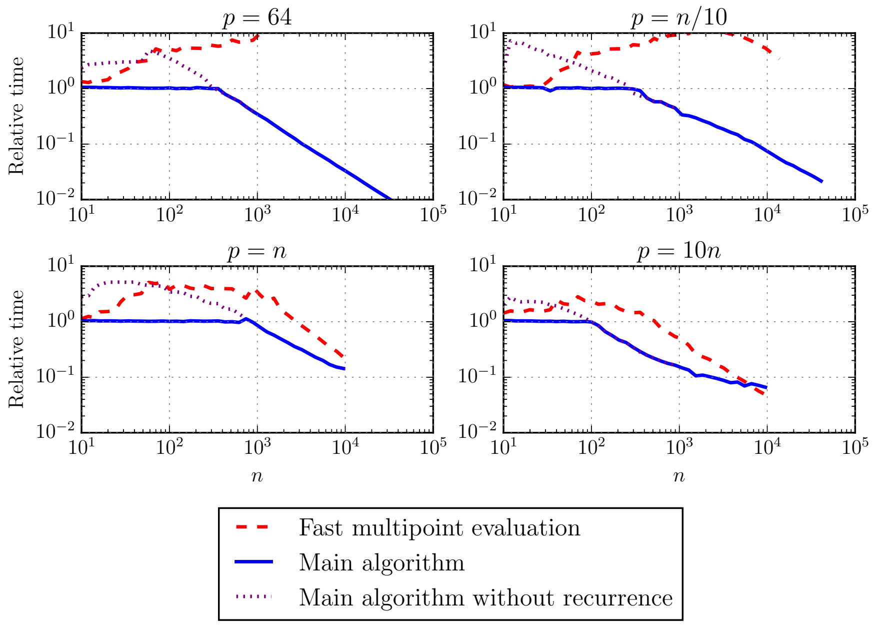

FIG. 2. *Performance comparison of various methods to evaluate $(P_n(x), P_n'(x))$ to p-bit precision for a set of $n/2$ points $0 < x < 1$ distributed like the roots of P_n. The y axis (relative time) shows the time divided by the time using the three-term recurrence in fixed-point arithmetic.*

Except where otherwise noted, we ran the programs under 64-bit Linux on a laptop with a 1.90 GHz Intel Core i5-4300U CPU using a single core.

7.1. Polynomial evaluation. Figure 2 compares the performance of different methods for evaluating $(P_n(x), P_n'(x))$ on a set of $n/2$ points distributed like the positive roots of $P_n(x)$ to simulate one stage of Newton iteration at p-bit precision. The time for the three-term recurrence is set to 1; i.e., we divide the other timings by this measurement. The following methods are timed:

- Our hybrid method (from here on called the "main algorithm") with automatic selection between the three-term recurrence and different series expansions.

- The main algorithm without the three-term recurrence as the basecase, i.e., using series expansions even for very small n.
- Fast multipoint evaluation of the expanded polynomials $P_n(x)$ and $P_n'(x)$. As in subsection 4.2, we expand $P_n(\sqrt{x})$ for even n and $P_n(\sqrt{x})/\sqrt{x}$ for odd n and evaluate at x^2 since this halves the amount of work. The polynomial coefficients are generated using the hypergeometric recurrence, and the fast multipoint evaluation is done using `_arb_poly_evaluate_vec_fast_precomp` (where the "precomp" suffix indicates that the same product tree is used for both P_n and P_n'). The fast multipoint evaluation is done with $2.9n$ guard bits, which was found experimentally to be sufficient for full accuracy.

The crossover point between the three-term recurrence and series expansions usually occurs around $n \approx 10^2 - 10^3$ (it can be as low as $n \approx 10$ if p is much larger). For modest n, the three-term recurrence is much faster than the hypergeometric series (typically by a factor 3–4) due to working with negligible extra precision and thanks to the low overhead of fixed-point arithmetic. This low overhead is very useful for typical evaluation of Legendre polynomials and generation of quadrature nodes for one or a few machine words of precision. The crossover point could be lowered slightly if we used a similarly optimized fixed-point implementation for the hypergeometric series.

When p is fixed (top left plot in Figure 2), the main algorithm is a factor $\mathcal{O}(n)$ faster than the three-term recurrence since the asymptotic expansion converges to sufficient accuracy after $\mathcal{O}(1)$ terms for all sufficiently large n. With the constant precision $p = 64$, the main algorithm is 3.0 times faster for $n = 10^3$ and 30 times faster for $n = 10^4$. Conversely, fast multipoint evaluation with constant p is a factor $\mathcal{O}(n)$ slower than our algorithm due to the higher internal precision.

When $p \propto n$ (the three remaining plots in Figure 2), the main algorithm appears to show the same $\mathcal{O}(n)$ speedup over the three-term recurrence after the crossover point, at least initially. This speedup should level off asymptotically, but in practice this only occurs for n larger than 10^4 where we have already gained a factor 10 or more. The leveling off is visible in the bottom right figure ($p = 10n$).

Fast multipoint evaluation gives a true asymptotic $\mathcal{O}(n)$ speedup, but since it has much higher overhead, it only starts to give an improvement over the main algorithm from $n \approx 10^4$ and for p larger than n. When $p = n/10$, it appears that fast multipoint evaluation will only break even for n much larger than 10^5. We conclude that fast multipoint evaluation would be worthwhile only for the last few Newton iterations when computing quadrature nodes for exceptionally high precision. Since independent evaluations are more convenient and easy to parallelize, the fast multipoint evaluation method currently seems to have limited practical value for this application.

7.2. Quadrature nodes. The function `arb_hypgeom_legendre_p_ui_root(x, w, n, k, p)` sets the output variable x to a ball containing the root of P_n with index k (we use the indexing $0 \le k < n$, with $k = 0$ giving the root closest to 1) computed to p-bit precision. It also sets w to the corresponding quadrature weight. We use the formulae in [27, Theorem 1(c)] to compute an initial enclosure with roughly machine precision, followed by refinements with the interval Newton method at doubling precision steps for very high precision. We deduce the quadrature weights thanks to the classical expression as functions of the nodes recalled in (3).

Table 1 shows timings for computing degree-n Gauss–Legendre rules to p-bit precision by calling this function repeatedly with $0 \le k < n/2$. Table 2 compares our code to the `intnumgaussinit` function in Pari/GP which uses a generic polynomial

Table 1

Time in seconds to compute the degree-n Gauss–Legendre rules with p-bit precision using our code. (For large n and p, the time was estimated by computing a subset of the nodes and weights.)

$n \setminus p$	64	256	1 024	3 333	33 333
20	0.000133	0.000229	0.000510	0.00121	0.0198
50	0.000450	0.000870	0.00212	0.00520	0.0710
100	0.00138	0.00310	0.00720	0.0163	0.191
200	0.00550	0.0111	0.0267	0.0550	0.589
500	0.0236	0.0610	0.164	0.325	2.61
1 000	0.0530	0.145	0.584	1.238	9.21
2 000	0.0860	0.298	1.12	4.20	32.6
5 000	0.191	0.665	2.67	14.3	181
10 000	0.350	1.26	4.93	26.6	674
100 000	3.60	12.2	41.3	212	13 637
1 000 000	58.0	146	411	1 850	103 960

Table 2

Time in seconds for Pari/GP to compute the degree-n Gauss–Legendre quadrature rules with p-bit precision. The numbers in parentheses show the speedup of our code compared to Pari/GP.

$n \setminus p$	64	256	1 024	3 333	33 333
20	0.00059 (×4.4)	0.00070 (×3.1)	0.0015 (×2.9)	0.0035 (×2.9)	0.078 (×3.9)
50	0.0043 (×9.6)	0.0050 (×5.8)	0.010 (×4.9)	0.022 (×4.1)	0.43 (×6.0)
100	0.020 (×15)	0.023 (×7.4)	0.045 (×6.3)	0.089 (×5.5)	1.5 (×8.1)
200	0.11 (×20)	0.13 (×11)	0.24 (×9.1)	0.45 (×8.1)	6.5 (×11)
500	2.0 (×86)	2.1 (×35)	2.9 (×18)	5.1 (×16)	58 (×22)
1 000	25 (×477)	26 (×180)	29 (×49)	39 (×32)	300 (×33)
2 000	496 (×5 767)	478 (×1 604)	474 (×423)	532 (×127)	1 880 (×58)

root isolation strategy followed by Newton iteration for high precision refinement. The improvement is most dramatic for small p and large n, where we benefit from using asymptotic expansions, but we also obtain a consistent speedup for large p.

For low precision and large n, our implementation is about three orders of magnitude slower than the machine precision code by Bogaert [5] which is reported to compute the nodes and weights for $n = 10^6$ in 0.02 seconds on four cores. This difference is reasonable since we use arbitrary-precision arithmetic, compute rigorous error bounds, and evaluate the Legendre polynomials explicitly whereas Bogaert uses a more sophisticated asymptotic development for both the nodes and the weights.

We also note that we can compute 53-bit floating-point values with provably correct rounding in about the same time as the 64-bit values, using Ziv's strategy of increasing the precision. For a ball with relative radius just larger than 2^{-64}, there is less than a 1% probability that the correct 53-bit rounding cannot be determined, in which case that particular node can be recomputed with a few more bits.

Fousse [12] reports a few timings for smaller n and high precision obtained on a 2.40 GHz AMD Opteron 250 CPU. For example, $n = 80, p = 500$ takes 0.14 seconds (our implementation takes 0.029 seconds) and $n = 556, p = 5\,000$ takes 17 seconds (our implementation takes 0.53 seconds). Of course, these timings are not directly comparable since different CPUs were used.

The `mathinit` program included with version 2.2.19 of Bailey's ARPREC library generates Gauss–Legendre quadrature nodes using Newton iteration together with the three-term recurrence for evaluating Legendre polynomials [4, 2]. With default parameters, this program computes the rules of degree $n = 3 \cdot 2^{i+1}$ for $1 \le i \le 10$ at 3 408 bits of precision, intended as a precomputation for performing degree-adaptive

Table 3

Left columns: Time in seconds to generate 1 000*-digit quadrature rules for the degrees* n *used by ARPREC. Right columns: For three different integrals, the error* $|\int_{-1}^{1} f(x)\mathrm{d}x - \sum_{k=0}^{n-1} w_k f(x_k)|$ *of the degree-*n *quadrature rule, and the time to evaluate this degree-*n *approximation of the integral at* 1 000*-digit precision in Arb given the nodes and weights* (x_k, w_k).

n	ARPREC	Our code	$\int_{-1}^{1}\log(2+x)\mathrm{d}x$		$\int_{-1}^{1}\mathrm{Ai}(10x)\mathrm{d}x$		$\int_{-1}^{1}\Gamma(1+ix)\mathrm{d}x$	
			Error	Time	Error	Time	Error	Time
12	0.00520	0.000592	10^{-14}		10^{-1}		10^{-8}	
24	0.0189	0.00171	10^{-28}		10^{-9}		10^{-17}	
48	0.0629	0.00507	10^{-56}		10^{-34}		10^{-36}	
96	0.251	0.0163	10^{-111}		10^{-105}		10^{-73}	
192	0.974	0.0532	10^{-222}		10^{-284}	0.075	10^{-146}	
384	3.83	0.195	10^{-441}	0.023	10^{-721}	0.15	10^{-293}	1.3
768	15.2	0.763	10^{-881}	0.045	$< \varepsilon$	0.29	10^{-588}	2.5
1 536	60.9	2.82	$< \varepsilon$	0.091			$< \varepsilon$	5.0
3 072	241	9.55						
6 144	1 013	18.3						

Table 4

Left columns: Step sizes h*, number of evaluation points, and time to compute nodes for double exponential quadrature with Arb at* 1 000*-digit precision. Right columns: Error and evaluation time given precomputed nodes.*

h	$2n+1$	Time	$\int_{-1}^{1}\log(2+x)\mathrm{d}x$		$\int_{-1}^{1}\mathrm{Ai}(10x)\mathrm{d}x$		$\int_{-1}^{1}\Gamma(1+ix)\mathrm{d}x$	
			Error	Time	Error	Time	Error	Time
2^{-7}	1 989	0.07	10^{-407}	0.12	10^{-423}	0.93	10^{-314}	6.3
2^{-8}	3 977	0.14	10^{-814}	0.25	10^{-909}	1.75	10^{-630}	13.0
2^{-9}	7 955	0.27	$< \varepsilon$	0.55	$< \varepsilon$	3.49	$< \varepsilon$	25.1

numerical integrations with up to 1 000 decimal digit accuracy. This takes about 1 300 seconds in total (our implementation takes 32 seconds). A breakdown for each degree level is shown in Table 3.

Table 3 also shows the approximation error and the evaluation time (not counting the computation of the nodes and weights) for the degree-n approximations of three different integrals, illustrating the relative costs and realistic requirements for n. As motivation for the third integral, we might think of a segment of a Mellin–Barnes integral. The log, Airy, and gamma function implementations in Arb are used.

The last few degree levels (with n roughly larger than the number of decimal digits) used by ARPREC tend to be dispensable for well-behaved integrands. A larger n is needed if the path of integration is close to a singularity or if the integrand is highly oscillatory. In such cases, bisecting the interval a few times to reduce the necessary n is often a better tradeoff. On the other hand, since the time to generate nodes with our code only grows linearly with n beyond $n \approx p$, increasing the degree further is viable, and potentially useful if the integrand is expensive to evaluate.

In the present work, we refrain from a more detailed discussion of adaptive integration strategies and the computation of error bounds for the integral itself. However, we mention that Arb contains an implementation of a version of the Petras algorithm [28] for rigorous integration. This code uses both adaptive path subdivision and Gauss–Legendre quadrature with an adaptive choice of n up to $n \approx 0.5p$ by default, with degree increments $n \approx 2^{k/2}$ and automatic caching of the nodes for fast repeated integrations. Node generation takes at most a few seconds for a first integration at 1 000-digit precision and a few milliseconds for 100-digit precision.

8. Gauss–Legendre versus Clenshaw–Curtis and the double exponential method.

The Clenshaw–Curtis and double exponential (tanh-sinh) quadrature schemes have received much attention as alternatives to Gauss–Legendre quadrature for numerical integration with very high precision [30, 2, 33]. Both schemes typically require a constant factor more evaluation points than Gauss–Legendre rules for equivalent accuracy, but the nodes and weights are easier to compute. Gauss–Legendre quadrature is therefore the best choice when the integrand is expensive to evaluate or when nodes can be precomputed for several integrations. It is of some interest to compare the relative costs empirically.

Here we assume an analytic integrand with singularities well isolated from the finite path of integration so that Gauss–Legendre quadrature is a good choice to begin with. As observed in [33], Clenshaw–Curtis often converges with identical rate to Gauss–Legendre for less well-behaved integrands, and the double exponential method is far superior to both Clenshaw–Curtis and Gauss–Legendre for analytic integrands with endpoint singularities.

Clenshaw–Curtis quadrature uses the Chebyshev nodes $\cos(\pi k/n), 0 \le k \le n$, and the corresponding weights can be expressed by a discrete cosine transform which takes $\mathcal{O}(n \log n)$ arithmetic operations to compute by an FFT. As a rule of thumb, $2n$-point Clenshaw–Curtis quadrature gives the same accuracy as n-point Gauss–Legendre quadrature (for instance, the 384-point Clenshaw–Curtis rule gives errors of 10^{-229}, 10^{-294}, and 10^{-154} for the three integrals in Table 3). As a point of comparison with Tables 1 and 3, Arb computes a length-2 048 FFT with 1 000-digit precision in 0.09 seconds and a length-32 768 FFT with 10 000-digit precision in 36 seconds. The precomputation for Clenshaw–Curtis quadrature is therefore roughly a factor 20 cheaper than that for Gauss–Legendre quadrature with our algorithm, while subsequent integration with cached weights is twice as expensive for Clenshaw–Curtis.

Double exponential quadrature uses the change of variables $x = \tanh(\frac{1}{2}\pi \sinh t)$ to convert an integral on $(-1, 1)$ to the interval $(-\infty, +\infty)$ in such a way that the trapezoidal rule $\int_{-\infty}^{\infty} f(t)dt \approx h \sum_{k=-n}^{n} f(hk)$ converges exponentially fast. One generally chooses the discretization parameter as $h = 2^{-j}$ so that both the evaluation points and weights can be recycled for successive levels $j = 1, 2, 3 \ldots$, and n is chosen so that the tail of the infinite series is smaller than 2^{-p}. The $2n + 1$ nodes and weights can be computed with $n + \mathcal{O}(1)$ exponential function evaluations and $\mathcal{O}(n)$ arithmetic operations. Double exponential quadrature with Cn evaluation points typically achieves the same accuracy as n-point Gauss–Legendre quadrature, where C is slightly larger than for Clenshaw–Curtis, e.g., $C \approx 5$; see Table 4. The time to compute nodes and weights is comparable to Clenshaw–Curtis quadrature (around 0.2 seconds for 1 000-digit precision and two minutes for 10 000-digit precision).

In summary, for integration with precision in the neighborhood of 10^3 to 10^4 digits, computing n nodes with our algorithm is about an order of magnitude more expensive than performing n elementary function (e.g., exp or log) evaluations or computing an FFT of length n. This makes Gauss–Legendre quadrature competitive for computing more than m integrals (assuming that nodes and weights are cached), for a single integral requiring splitting into m subintervals, or for a single integral when the integrand costs more than m elementary function evaluations, where $m \approx 10^1$.

The picture becomes more complicated when accounting for the method used to estimate errors in an adaptive integration algorithm. One drawback of the Gauss–Legendre scheme is that the nodes are not nested, so that an adaptive strategy that repeatedly doubles the quadrature degree requires twice as many function evaluations as the degree of the final level. This drawback disappears if the error is estimated by extrapolation or if an error bound is computed a priori as in the Petras algorithm [28].

9. Conclusion. In [2], it was claimed that "There is no known scheme for generating Gaussian abscissa–weight pairs that avoids [the] quadratic dependence on n. High-precision abscissas and weights, once computed, may be stored for future use. But for truly extreme-precision calculations—i.e., several thousand digits or more—the cost of computing them even once becomes prohibitive."

In this quote, "quadratic dependence" refers to the number of arithmetic operations. We may remark that using asymptotic expansions in the evaluation of Legendre polynomials avoids the quadratic dependence on n for fixed precision p, and Theorem 1 avoids the implied cubic time dependence on n when $n = \mathcal{O}(p)$. In fact, Theorem 1 implies that Gauss–Legendre, Clenshaw–Curtis, and double exponential quadrature have the same quasi-optimal asymptotic bit complexity, up to logarithmic factors.

Our experiments show that the algorithm in Theorem 1 is hardly worthwhile. However, the hybrid method described in sections 2–5 does achieve a significant speedup for practical p and n which allows us to compute Gauss–Legendre quadrature rules for 1 000-digit integration in 1–2 seconds and for 10 000-digit integration in 10–20 minutes on a single core. This is not prohibitively expensive compared to the repeated evaluation of typical integrands, especially if several integrations are needed. Parallelization is also trivial since all roots are computed independently.

A natural extension of this work would be to consider Gaussian quadrature rules for different weight functions. The techniques should transfer to other classical orthogonal polynomials (Jacobi, Hermite, Laguerre, etc.) which likewise have hypergeometric expansions and satisfy three-term recurrence relations. The main obstacle might be to obtain large-n asymptotic expansions with suitable error bounds.

Acknowledgments. We are indebted to Nick Trefethen and two anonymous referees for their useful comments. We also thank Bruno Salvy for pointing out Petras's paper [27].

REFERENCES

[1] V. Antonov and K. Holševnikov, *An estimate of the remainder in the expansion of the generating function for the Legendre polynomials (generalization and improvement of Bernstein's inequality)*, Vestnik Leningrad Univ. Mat., 13 (1981), pp. 163–166. Translated by H. H. McFadden.

[2] D. H. Bailey and J. M. Borwein, *High-precision numerical integration: Progress and challenges*, J. Symbolic Comput., 46 (2011), pp. 741–754, https://doi.org/10.1016/j.jsc.2010.08.010.

[3] D. H. Bailey, J. M. Borwein, and R. E. Crandall, *Integrals of the Ising class*, J. Phys. A, 39 (2006), 12271, https://doi.org/10.1088/0305-4470/39/40/001.

[4] D. H. Bailey, H. Yozo, X. S. Li, and B. Thompson, *ARPREC: An Arbitrary Precision Computation Package*, Technical report LBNL-53651, Lawrence Berkeley National Lab, Berkely, CA, 2002.

[5] I. Bogaert, *Iteration-free computation of Gauss–Legendre quadrature nodes and weights*, SIAM J. Sci. Comput., 36 (2014), pp. A1008–A1026, https://doi.org/10.1137/140954969.

[6] I. Bogaert, B. Michiels, and J. Fostier, $\mathcal{O}(1)$ *computation of Legendre polynomials and Gauss–Legendre nodes and weights for parallel computing*, SIAM J. Sci. Comput., 34 (2012), pp. C83–C101, https://doi.org/10.1137/110855442.

[7] R. P. Brent, *Asymptotic Approximation of Central Binomial Coefficients with Rigorous Error Bounds*, preprint, https://arxiv.org/abs/1608.04834, 2016.

[8] R. P. Brent and P. Zimmermann, *Modern Computer Arithmetic*, Cambridge University Press, Cambridge, UK, 2010, http://www.loria.fr/~zimmerma/mca/mca-cup-0.5.7.pdf.

[9] D. Broadhurst, *Feynman Integrals, L-Series and Kloosterman Moments*, preprint, https://arxiv.org/abs/1604.03057, 2016.

[10] Y. Chow, L. Gatteschi, and R. Wong, *A Bernstein-type inequality for the Jacobi polynomial*, Proc. Amer. Math. Soc., 121 (1994), pp. 703–709, https://doi.org/10.2307/2160265.

[11] D. V. Chudnovsky and G. V. Chudnovsky, *Computer algebra in the service of mathematical physics and number theory*, in Computers in Mathematics (Stanford, CA, 1986), D. V. Chudnovsky and R. D. Jenks, eds., Lecture Notes in Pure and Appl. Math. 125, Dekker, New York, 1990, pp. 109–232.

[12] L. Fousse, *Accurate multiple-precision Gauss–Legendre quadrature*, in 18th IEEE Symposium on Computer Arithmetic, ARITH'07, IEEE, Washington, DC, 2007, pp. 150–160.

[13] G. H. Golub and J. H. Welsch, *Calculation of Gauss quadrature rules*, Math. Comp., 23 (1969), pp. 221–230, https://doi.org/10.1090/S0025-5718-69-99647-1.

[14] T. Granlund and the GMP development team, *GNU MP: The GNU Multiple Precision Arithmetic Library*, 6.1.2 ed., 2017, https://gmplib.org/.

[15] N. Hale and A. Townsend, *Fast and accurate computation of Gauss–Legendre and Gauss–Jacobi quadrature nodes and weights*, SIAM J. Sci. Comput., 35 (2013), pp. A652–A674, https://doi.org/10.1137/120889873.

[16] F. Johansson, *Evaluating parametric holonomic sequences using rectangular splitting*, in Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation, ISSAC '14, ACM, New York, 2014, pp. 256–263, https://doi.org/10.1145/2608628.2608629.

[17] F. Johansson, *Efficient implementation of elementary functions in the medium-precision range*, in 22nd IEEE Symposium on Computer Arithmetic, ARITH22, 2015, pp. 83–89, https://doi.org/10.1109/ARITH.2015.16.

[18] F. Johansson, *Computing Hypergeometric Functions Rigorously*, preprint, https://arxiv.org/abs/1606.06977, 2016.

[19] F. Johansson, *Arb: Efficient arbitrary-precision midpoint-radius interval arithmetic*, IEEE Trans. Comput., 66 (2017), pp. 1281–1292, https://doi.org/10.1109/TC.2017.2690633.

[20] F. Johansson and I. V. Blagouchine, *Computing Stieltjes Constants Using Complex Integration*, preprint, https://arxiv.org/abs/1804.01679, 2018.

[21] A. Kobel and M. Sagraloff, *Fast Approximate Polynomial Multipoint Evaluation and Applications*, preprint, https://arxiv.org/abs/1304.8069, 2013.

[22] M. A. Kowalski, A. G. Werschulz, and H. Woźniakowski, *Is Gauss quadrature optimal for analytic functions?*, Numer. Math., 47 (1985), pp. 89–98, https://doi.org/10.1007/BF01389877.

[23] P. Molin, *Numerical Integration and L-Functions Computations*, theses, Université Sciences et Technologies - Bordeaux 1, Talence, France, 2010.

[24] R. E. Moore, *Methods and Applications of Interval Analysis*, SIAM, Philadelphia, 1979, https://doi.org/10.1137/1.9781611970906.

[25] F. W. J. Olver, *Asymptotics and Special Functions*, A K Peters, Wellesley, MA, 1997.

[26] F. W. J. Olver, D. W. Lozier, R. F. Boisvert, and C. W. Clark, *NIST Handbook of Mathematical Functions*, Cambridge University Press, New York, 2010.

[27] K. Petras, *On the computation of the Gauss–Legendre quadrature formula with a given precision*, J. Comput. Appl. Math., 112 (1999), pp. 253–267, https://doi.org/10.1016/S0377-0427(99)00225-3.

[28] K. Petras, *Self-validating integration and approximation of piecewise analytic functions*, J. Comput. Appl. Math., 145 (2002), pp. 345–359, https://doi.org/10.1016/S0377-0427(01)00586-6.

[29] D. M. Smith, *Efficient multiple-precision evaluation of elementary functions*, Math. Comp., 52 (1989), pp. 131–134, http://myweb.lmu.edu/dmsmith/MComp1989.pdf.

[30] H. Takahasi and M. Mori, *Double exponential formulas for numerical integration*, Publ. Res. Inst. Math. Sci., 9 (1974), pp. 721–741, https://doi.org/10.2977/prims/1195192451.

[31] The PARI Group, *PARI/GP version* 2.9.4, Univ. Bordeaux, 2017, http://pari.math.u-bordeaux.fr/.

[32] A. Townsend, *The race for high order Gauss–Legendre quadrature*, SIAM News, 48 (2015), pp. 1–3, http://math.mit.edu/~ajt/papers/QuadratureEssay.pdf.

[33] L. N. Trefethen, *Is Gauss quadrature better than Clenshaw–Curtis?*, SIAM Rev., 50 (2008), pp. 67–87, https://doi.org/10.1137/060659831.

[34] L. N. Trefethen, *Six Myths of Polynomial Interpolation and Quadrature*, https://people.maths.ox.ac.uk/trefethen/mythspaper.pdf, 2011.

[35] J. van der Hoeven, *Ball Arithmetic*, Technical report, HAL, 2009, http://hal.archives-ouvertes.fr/hal-00432152/fr/.

[36] J. Wimp, *Computation with Recurrence Relations*, Pitman, Boston, 1984.