FAST AND ACCURATE EVALUATION OF WIGNER $3j$, $6j$, AND $9j$ SYMBOLS USING PRIME FACTORIZATION AND MULTIWORD INTEGER ARITHMETIC*

H. T. JOHANSSON† AND C. FORSSÉN‡

Abstract. We present an efficient implementation for the evaluation of Wigner $3j$, $6j$, and $9j$ symbols. These represent numerical transformation coefficients that are used in the quantum theory of angular momentum. They can be expressed as sums and square roots of ratios of integers. The integers can be very large due to factorials. We avoid numerical precision loss due to cancellation through the use of multiword integer arithmetic for exact accumulation of all sums. A fixed relative accuracy is maintained as the limited number of floating-point operations in the final step incur rounding errors only in the least significant bits. Time spent to evaluate large multiword integers is in turn reduced by using explicit prime factorization of the ingoing factorials, thereby improving execution speed. Comparison with existing routines shows the efficiency of our approach, and we therefore provide a computer code based on this work.

Key words. computational physics, prime factorization, multiword integer arithmetic, quantum theory of angular momentum

AMS subject classifications. 81-04, 81Rxx, 40-04

DOI. 10.1137/15M1021908

1. Introduction. Wigner $3j$, $6j$, and $9j$ symbols (xj symbols) are used in physics and chemistry whenever one deals with angular momenta in quantum mechanical systems [20, 7]. These functions correspond to transformation coefficients between different spin representations. The input arguments are six or nine integers, or half integers, that represent different angular momenta j and projection quantum numbers $|m| \leq j$. Because of their frequent appearance in scientific computing, standard library functions that evaluate Wigner symbols are available in several mathematics software systems [18, 1, 2] and computational libraries [11]. In fact, the quantum spin properties of electrons, protons, neutrons, and atoms imply that angular momentum recoupling is a general and important ingredient in the theoretical modeling of quantum many-body systems [10]. Well-known examples are found in quantum chemistry [6], condensed matter [8], and atomic [12] and nuclear physics [17]. More recently, the mathematical apparatus of quantum-mechanical angular momentum coupling has found a prominent role also in quantum gravity and quantum computing applications [5].

In particular, the Wigner symbols enter at a critical stage in the computational

†Department of Fundamental Physics, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (f96hajo@chalmers.se).
‡Department of Fundamental Physics, Chalmers University of Technology, SE-412 96 Göteborg, Sweden; Department of Physics and Astronomy, University of Tennessee, Knoxville, TN 37996; and Physics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831 (christian.forssen@chalmers.se).

chain of quantum many-body modeling, namely when calculating the strength of the interaction between pairs or triples of particles. The evaluation of these so-called interaction matrix elements can involve intermediate angular-momentum sums that extend to very large values ($j \sim 100$). Eventually, a sum over all possible particle pairs or triples within the many-body system gives the total interaction energy. The continued efforts to improve accurate calculations [16, 22, 4], constructing recursive relations [15, 13, 19], and achieving fast lookup [14] show the central place of these symbols in computations.

Accurate evaluation of Wigner symbols with large j using floating-point arithmetic is difficult because of cancellation in sums of large alternating terms. The precision losses due to cancellations can be avoided by using integer arithmetic. However, this approach generally involves very large numbers due to factorials, thus requiring the use of multiword integer representations in computer codes. Multiword integers are multiple machine words used together to represent arbitrarily large numbers. A clever reduction of the huge integers, by expressing the terms as products of binomial coefficients, was demonstrated by Wei [22]. In this context, one can also mention early efforts in introducing prime factorization [9, 16]. However, high-accuracy approaches are in general significantly slower than straightforward floating-point implementations. As an example, the computation of $6j$ symbols for small angular momenta, $\max(j) = 3$, is already an order of magnitude slower when using the published code of [22] as compared to a floating-point calculation with the popular GNU Scientific Library (GSL) [11]. This difference grows with increasing j.

In this paper we present a new method in which the big-integer penalties are mitigated by making explicit use of prime-factorized factorials. In our tests this method executes considerably faster than previous high-accuracy approaches, provides a bounded and fixed relative accuracy, and extends to very large values of ingoing angular momenta.

2. Method. The implementation uses the fact that the xj symbols can be written in the form

$$(2.1) \qquad W_x = \frac{n\sqrt{s}}{q},$$

where n, s, and q are integers. This general expression comes directly from the defining equations [20]. For example, for $6j$ symbols we have the Racah formula

$$(2.2) \qquad \left\{ \begin{array}{ccc} a & b & c \\ d & e & f \end{array} \right\} = \prod_{i=1}^{4} \sqrt{\Delta(\hat{\alpha}_i)} \sum_k (-1)^k \frac{(k+1)!}{\prod_{i=1}^{4}(k-\alpha_i)! \prod_{j=1}^{3}(\beta_j - k)!},$$

with

$$(2.3) \qquad \Delta(a,b,c) = \frac{(a+b-c)!(a-b+c)!(b+c-a)!}{(a+b+c+1)!}$$

and

$$(2.4) \qquad \begin{array}{lll} \hat{\alpha}_1 = (a,b,c), & \alpha_1 = a+b+c, & \beta_1 = a+b+d+e, \\ \hat{\alpha}_2 = (d,e,c), & \alpha_2 = d+e+c, & \beta_2 = a+c+d+f, \\ \hat{\alpha}_3 = (a,e,f), & \alpha_3 = a+e+f, & \beta_3 = b+c+e+f, \\ \hat{\alpha}_4 = (d,b,f), & \alpha_4 = d+b+f. \end{array}$$

The summation index k is in the range $\max(\alpha_1, \alpha_2, \alpha_3, \alpha_4) \le k \le \min(\beta_1, \beta_2, \beta_3)$. As stated before, the input arguments $a, b, c, d, e,$ and f are integers or half-integers. For

TABLE 1
Evaluation of the 6j symbol {2 2 2; 2 2 2} that illustrates the identification of the common factor (including the LCD) and the accumulation of the numerator sum. The fifth column corresponds to the tuple representation of each term, i.e., the numerator tuple subtracted by the denominator product tuple. See text for details.

Term	Sign	Numerator	Denominator	Total ratio (prime tuple)	Final numerator (prime tuple)	(integer)
$k = 6$	$+1$	$7!$	$2!2!2!$	$(1, 2, 1, 1)$	$(0, 0, 0, 0)$	1
$k = 7$	-1	$8!$	1	$(7, 2, 1, 1)$	$(6, 0, 0, 0)$	-64
$k = 8$	$+1$	$9!$	$2!2!2!2!$	$(3, 4, 1, 1)$	$(2, 2, 0, 0)$	36
		Common factor/LCD:		$(1, 2, 1, 1)$	Sum:	-27

a symbol not to be trivially zero, the sums α_i must be integer, and $\hat{\alpha}_i$ must fulfill triangle conditions, e.g., for $\hat{\alpha}_1$:

$$(2.5) \qquad\qquad |a - b| \le c \le a + b.$$

The overall structure for $3j$ symbols is similar. Note how all expressions contain (factorials of) integers. While being possibly large, they can be evaluated exactly. By finding and using the least common denominator (LCD) of the sum, expression (2.2) can be brought to the form (2.1).

Finding the LCD is straightforward by expressing each factorial as an explicit product of prime numbers, e.g., $7! = 2^4 \times 3^2 \times 5^1 \times 7^1$, treated as the tuple $(4, 2, 1, 1)$ in the calculations. The tuple representing each term is found by subtracting the exponents related to each prime number in all factors of the denominator from those representing the numerator. The LCD is then found by taking the minimum value for each exponent over all terms. In practice, the LCD is reduced considerably since the numerator is also included in this procedure. Factors common to all terms are then represented by positive exponents. As an explicit example we consider the computation of the $6j$ symbol $\{ \begin{smallmatrix} 2 & 2 & 2 \\ 2 & 2 & 2 \end{smallmatrix} \}$. The sum that needs to be evaluated for this symbol includes three terms; see Table 1. The final common factor of the sum (which includes the LCD) is found to have the tuple representation $(1, 2, 1, 1)$. The positive exponents are a consequence of the relatively large numerators appearing in the sum.

The common factor is then subtracted from the exponent tuples of each term, giving pure numerators. In Table 1 this step corresponds to the second-to-last column. They are converted into exact integers using multiword arithmetic. The sum is then easily accumulated, yielding one part of n in (2.1).

The product of the Δ-factors is formed by summing the factorial prime exponent contributions. As it shall suffer a square root, one factor will be removed from each odd exponent to make sure that all remaining ones are even. The removed odd ones will build the term $\sqrt{s}$ in (2.1), while the remaining even ones will be divided by 2, thereby evaluating the square root. In our example, the product under the square root becomes $\frac{2!^3}{7!} \frac{2!^3}{7!} \frac{2!^3}{7!} \frac{2!^3}{7!}$, which is represented by the tuple $(-1, -2, -1, -1)^4 = (-4, -8, -4, -4)$. All exponents are even, so applying the square root simply gives $(-2, -4, -2, -2)\sqrt{1}$.

The common factor is then added, giving final exponents that are used to create q and the second part of n in (2.1), by using the negative and positive exponents, respectively. In our example, the product of the Δ-factors is added to the common factor of Table 1, giving the final exponents $(-1, -2, -1, -1)$ that together correspond to $1/630$. The two parts of n are combined using multiword integer multiplication.

In the final step, the value of W_x in (2.1) can be evaluated using floating-point arithmetic with a limited loss in relative precision. In our example, this corresponds to the evaluation of $-27\sqrt{1}/630 = -0.0429$. The three conversions from integer to floating point, together with one square root, one multiplication, and one division, constitute six operations, each of which incurs the loss of (at most) half a least significant bit in the floating-point representation used, commonly referred to as the machine epsilon, ϵ. For the 64-bit floating-point format of IEEE 758, which is usually used for the C type `double`, one has $\epsilon = 1.11 \cdot 10^{-16}$. This implies a maximum relative error of $6\epsilon = 6.66 \cdot 10^{-16}$.

2.1. Wigner $9j$. Wigner $9j$ symbols can be written as a sum of products of $6j$ symbols; see, e.g., [20]. It is not obvious from this direct formula for Wigner $9j$ symbols that they can be written in the form (2.1) due to the square roots in the Δ-prefactors of the $6j$ symbols. Nevertheless, a rearranged formula based on binomials was presented [21] and used [22] by Wei. This formula is also employed in the present implementation, with the small modification of expressing the binomials as factorials. In total, the $9j$ evaluation becomes a double sum where an overall LCD expressed in prime-exponent form can be extracted. The routine implementing the sum in the $6j$ formula is reused for this purpose.

3. Results. Calculations using the `wigxjpf` routine presented in this work have been compared (in floating point) with the `369j` code by Wei [22] and with a simple implementation in which we use floating-point operations from precalculated factorial tables (see section 4). The floating-point based routines of GSL (version 1.16) [11] have also been included in the measurements. The comparisons are in terms of both execution speed and accuracy. The benchmark calculations for measuring the accuracy are performed with our routine, `wigxjpf`, using extended precision computations (the 80-bit `long double` of the x87 floating-point unit, $\epsilon = 5.42 \cdot 10^{-20}$). This benchmark is reliable since only the last floating-point stage of the current routine introduces rounding errors.

Results are presented in Figures 1 and 2 for $3j$ and $(6j, 9j)$ symbols, respectively. For reasonable maximum j, all symbols not trivially zero by triangle conditions have been calculated (solid lines), which provides an exhaustive measurement of the average evaluation speed and the maximum relative error. However, for larger symbols the measurement is approximative as we have restricted the computations to a random subset of nontrivially zero symbols (dotted lines with markers).

In all measurements, the present work is slightly more accurate and up to an order of magnitude faster than `369j`. The relative error never exceeded $6\epsilon \approx 6.66 \cdot 10^{-16}$, which is our upper bound. The slightly larger accuracy loss in `369j` is due to individual conversion of each prime power to floating point and subsequent multiplication. The execution time difference between our implementation and the fast floating-point based routines is less than an order of magnitude at low j, showing that the `wigxjpf` routines presented in this work are also very efficient. In comparison to GSL (version 1.16), `wigxjpf` is actually faster for $3j$ symbols up to $\max(j) = 60$, while the difference in execution speed always remains smaller than a factor 4 up to $\max(j) = 20$ (30) for $6j$ ($9j$) symbols. At large j the floating-point routines obviously suffer from an increasing loss of accuracy, thus making `wigxjpf` superior.

Explicit numerical values, execution times, and memory usage for a selected set of symbols are presented in Table 2. Note that we present results for extreme cases such as $j = 50,000$ and $j = 2,000$ for $6j$ and $9j$ symbols, respectively.
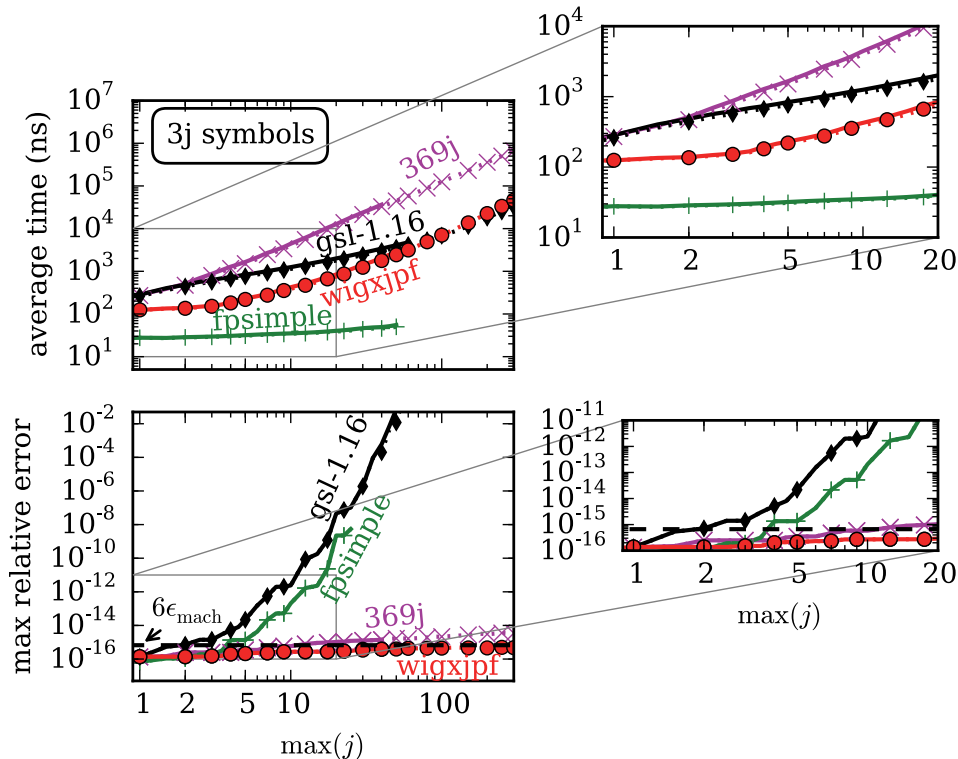
FIG. 1. *Upper panel: Average evaluation time for 3j symbols using four different implementations:* `wigxjpf` *(this work, circle markers),* `369j` *([22], crosses), GSL-1.16 ([11], diamond markers), and a simple floating-point computation (plus markers). The solid lines correspond to the exhaustive measurements where all nontrivially zero symbols with a specific maximum j have been computed. The dotted lines represent an approximation to the average evaluation time obtained from computations of a random subset. The overhead time to enumerate symbols, etc., has been subtracted. Lower panel: Maximum relative error, as compared to the symbols evaluated using* `long double` *on x86 hardware. All tests have been performed using a Xeon E3-1240v3, single-threaded at 3.8 GHz. The dashed line indicates the expected maximum relative error of our implementation, which is six times the machine epsilon.*

4. Implementation. Before the calculation of individual symbols, a table with precalculated factorizations of factorials is prepared. Table 3 shows the maximum possible factorial that has to be computed for evaluation of different symbols. The time spent on this initialization stage is amortized for applications that make repeated use of the routines. The table also gives the maximum number of terms in the sum (inner sum for 9j symbols). Temporary results (numerator-denominator exponent tuples) are stored for each term during evaluation of the sum.

First, all prime numbers up to the largest factorial argument, p, are determined by the sieve of Eratosthenes. By simple enumeration of integer exponent combinations in the basis of prime numbers, the factorization of all integers up to p are determined. Thus the factorizations are performed without any "test" divisions, i.e., checks for remainder zero. Each factorization is stored as an array of integers. The factorizations of factorials are constructed by cumulating the previous factorizations.

Multiplying (or dividing) two such factorizations is a matter of adding (or sub-
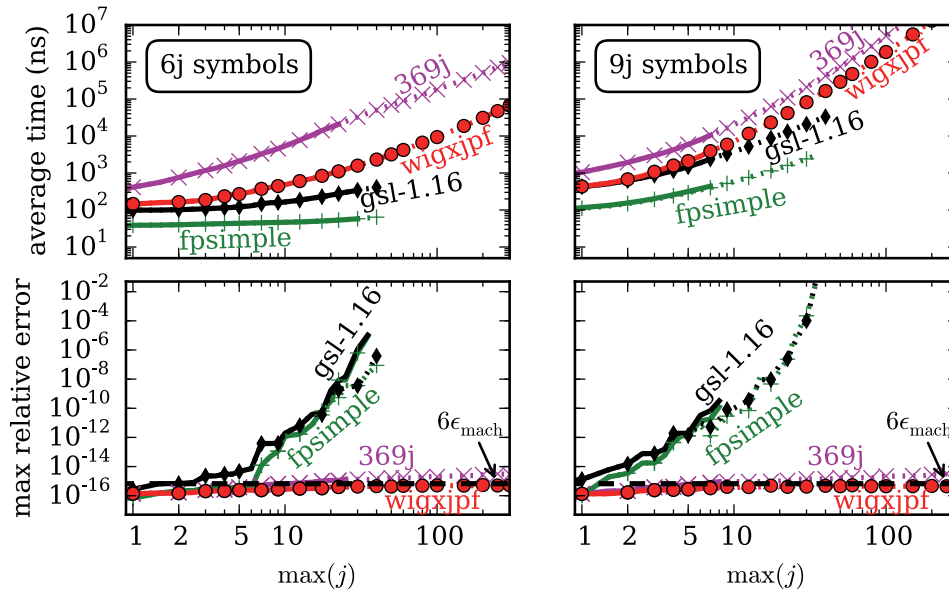
Fig. 2. *Same as Figure 1 but for the evaluation of* $6j$ *(left) and* $9j$ *(right) symbols.*

Table 2
Values and execution times for selected xj symbols. These times include setup of the precalculated tables of factorized factorials. The memory used for those tables and other temporary storage is also given. All tests have been performed using a Xeon E3-1240v3, single-threaded at 3.8 GHz.

	Symbol	Value	Time	Memory
$3j$	$\begin{pmatrix} 15 & 30 & 40 \\ 2 & 2 & -4 \end{pmatrix}$	-0.01908157979919155	0.07 ms	58 kB
	$\begin{pmatrix} 200 & 200 & 200 \\ -10 & 60 & -50 \end{pmatrix}$	0.0007493927313989515	0.31 ms	730 kB
	$\begin{pmatrix} 50,000 & 50,000 & 50,000 \\ 1,000 & -6,000 & 5,000 \end{pmatrix}$	$-1.116843916927519\text{e-}05$	112 s	19 GB
$6j$	$\begin{Bmatrix} 8 & 8 & 8 \\ 8 & 8 & 8 \end{Bmatrix}$	-0.01265208072315355	0.06 ms	7.0 kB
	{ all 200 }	0.0001559032124132416	0.62 ms	1.0 MB
	{ all 600 }	$-1.03981778344144\text{e-}07$	6.1 ms	8.0 MB
	{ all 10,000 }	$2.770313640470537\text{e-}08$	8.2 s	1.5 GB
	{ all 50,000 }	$3.997351841910046\text{e-}08$	816 s	32 GB
$9j$	$\begin{Bmatrix} 8.5 & 9.5 & 7.0 \\ 12.5 & 8.0 & 8.5 \\ 8.0 & 10.5 & 9.5 \end{Bmatrix}$	0.0002812983019125448	0.08 ms	20 kB
	$\begin{Bmatrix} 100 & 80 & 50 \\ 50 & 100 & 70 \\ 60 & 50 & 100 \end{Bmatrix}$	$1.055977980657612\text{e-}07$	1.9 ms	0.50 MB
	{ all 200 }	$1.278335300545066\text{e-}07$	0.15 s	1.6 MB
	{ all 1,000 }	$1.749851385596156\text{e-}09$	29.8 s	30 MB
	{ all 2,000 }	$2.755181565857189\text{e-}10$	358 s	109 MB

Maximum factorial argument and number of terms given the maximum j in a symbol. The limits can be rounded down when noninteger. These limits must be considered for temporary memory allocation. The memory requirement for the precalculated factorial table is given in the last columns for three different values of $\max(j)$.

			Memory requirement		
Symbol	Max factorial, $p!$	Max terms	$j = 100$	$j = 1{,}000$	$j = 10{,}000$
$3j$	$(3j + 1)!$	$j + 1$	193 kB	10.8 MB	0.78 GB
$6j$	$(4j + 1)!$	$j + 1$	309 kB	17.9 MB	1.35 GB
$9j$	$(5j + 1)!$	$j + 1$	450 kB	27.5 MB	2.06 GB

tracting) each element of the arrays, giving a new array. Adding or subtracting two factorizations would be more complicated, as the result, expressed as a factorization, could be completely different, even requiring much larger prime factors. To avoid this, each term is converted from the prime-factor array representation to multiword integer before addition or subtraction. No attempt is made to extract common factors between different multiword integers, as this is expensive and would have no impact on accuracy.

The multiword integers are just multiple machine words (32 or 64 bits) used to represent arbitrarily large numbers. The highest word is treated as having a sign bit, while all others are unsigned. Routines that perform addition, subtraction, and multiplication are implemented. The main difficulty is to propagate overflow (i.e., carry) between the words. The execution times for addition and subtraction are linear in the number of words used, while multiplication is quadratic.

The conversion of a number in prime-factor form to multiword integer form evaluates the contribution for each prime number separately, before multiplying them together. We utilize the fact that, for each exponent, each successively higher bit in its binary number representation corresponds to the square of the value represented by the previous bit. Therefore, two values are kept while iterating through the bits of each exponent: the repeatedly squared prime number and a cumulative product. While nonzero bits remain, the first number is squared each iteration, starting as the prime number for the first bit. For each bit set to 1, the cumulative product is multiplied by the hitherto squared result. These intermediate results are treated as multiword integers when necessary.

Conversion from multiword integer to floating point in the final stages is done word by word, from lower to higher words. The contributions that cannot be represented (i.e., that are below the precision) in the resulting floating-point variable are therefore just lost/ignored.

Repeated similar calculations of factorial arguments can be avoided by noting that the factors (and denominators) of each successive term in the sum depend on the summation index k in such a way that they always change their argument by 1 for each iteration. Therefore, factorial arguments (table addresses) can be calculated before the sum loop. In the actual loop these addresses are then just moved up or down one step each iteration.

Symbols that are trivially zero can be identified using a single `if`-clause in combination with two's complement arithmetic and bit manipulation. Only symbols passing this test are evaluated.

For the comparisons presented in section 3 we have also implemented simple floating-point routines. These routines use precalculated floating-point factorial val-

ues employing the same list-address technique for the sum as described above and thus become very fast. The factorials, stored as floating-point values, are calculated precisely using the multiword integers. The accuracy loss is completely dominated by cancellation in the sums of large alternating terms.

4.1. Memory usage. The largest memory requirement of `wigxjpf` is associated with the table of precalculated factorizations of numbers and factorials; see Table 3. The next largest use is in the temporary arrays of prime factors of each term in the sum. Other variables that are used during computations, e.g., multiword integers, are comparatively small in size.

5. Conclusions and outlook. In conclusion, we have successfully implemented a computational method for fast and accurate evaluation of Wigner $3j$, $6j$, and $9j$ symbols using prime factorization and multiword integer arithmetic. The efficiency and accuracy of our approach has been validated via benchmark calculations and comparison with existing floating-point routines and with the `369j` code by Wei [22]. The latter algorithm is similar to our approach in the sense that it also uses prime-number factorization and multiword integer arithmetic. However, it employs recursive calculation of binomial coefficients, while our implementation, with the precalculation of a table with prime-number factorizations, offers a code that is significantly faster—comparable in speed even to the floating-point routines. In addition, the precomputed factorization table makes our approach particularly amenable to applications in which multiple evaluations of angular momentum coupling coefficients are needed.

Furthermore, our implementation produces results with a maximum relative error of only six times the machine epsilon for $3j$, $6j$, and $9j$ symbols, also for very large angular momenta. In comparison, the error in the evaluation of $6j$ symbols using GSL 1.16 is five orders of magnitude larger already for $j = 15$ and quickly grows with increasing values of the angular momenta.

A computer code with the routines presented in this paper, `wigxjpf`, is available for download [3].

REFERENCES

[1] *Mathematica*, version 10.1 ed., Wolfram Research, Champaign, IL, 2015.
[2] *Sage Mathematics Software (Version 6.9)*, The Sage Developers, 2015, http://www.sagemath.org.
[3] `wigxjpf` *Computer Code*, http://fy.chalmers.se/subatom/wigxjpf.
[4] R. W. Anderson, V. Aquilanti, and F. C. da Silva, *Exact computation and large angular momentum asymptotics of 3nj symbols: Semiclassical disentangling of spin networks*, J. Chem. Phys., 129 (2008), 161101.
[5] V. Aquilanti, A. C. P. Bitencourt, C. S. da Ferreira, A. Marzuoli, and M. Ragni, *Quantum and semiclassical spin networks: From atomic and molecular physics to quantum computing and gravity*, Phys. Scripta, 78 (2008), 058103.
[6] P. W. Atkins and R. S. Friedman, *Molecular Quantum Mechanics*, Oxford University Press, Oxford, UK, 2011.
[7] D. M. Brink and G. R. Satchler, *Angular Momentum*, Oxford Sci. Publ., Clarendon Press, Oxford, UK, 1993.
[8] W. H. Dickhoff and D. Van Neck, *Many-Body Theory Exposed!: Propagator Description of Quantum Mechanics in Many-Body Systems*, World Scientific, Singapore, 2008.
[9] R. McD. Dodds and G. Wiechers, *Vector coupling coefficients as products of prime factors*, Comput. Phys. Comm., 4 (1972), pp. 268–274.

[10] A. L. Fetter and J. D. Walecka, *Quantum Theory of Many-Particle Systems*, Dover Books on Physics, Dover, Mineola, NY, 2003.

[11] M. Galassi et al., *GNU Scientific Library Reference Manual*, 3rd ed., 2009; available online from http://www.gnu.org/software/gsl.

[12] I. Lindgren and J. Morrison, *Atomic Many-Body Theory*, Springer Ser. Atomic, Optical, Plasma Phys., Springer-Verlag, Berlin, Heidelberg, 2012.

[13] J. Luscombe and M. Luban, *Simplified recursive algorithm for Wigner 3j and 6j symbols*, Phys. Rev. E, 57 (1998), pp. 7274–7277.

[14] J. Rasch and A. C. H. Yu, *Efficient storage scheme for precalculated Wigner 3j, 6j and Gaunt coefficients*, SIAM J. Sci. Comput., 25 (2003), pp. 1416–1428.

[15] K. Schulten and R. G. Gordon, *Exact recursive evaluation of 3j- and 6j-coefficients for quantum-mechanical coupling of angular momenta*, J. Math. Phys., 16 (1975), pp. 1961–1970.

[16] A. J. Stone and C. P. Wood, *Root-rational-fraction package for exact calculation of vector-coupling coefficients*, Comput. Phys. Comm., 21 (1980), pp. 195–205.

[17] J. Suhonen, *From Nucleons to Nucleus: Concepts of Microscopic Nuclear Theory*, Theoret. Math. Phys., Springer, Berlin, Heidelberg, 2007.

[18] SymPy Development Team, *SymPy: Python Library for Symbolic Mathematics*, http://www.sympy.org (2014).

[19] R. E. Tuzun, P. Burkhardt, and D. Secrest, *Accurate computation of individual and tables of 3-j and 6-j symbols*, Comput. Phys. Comm., 112 (1998), pp. 112–148.

[20] D. A. Varshalovich, A. N. Moskalev, and V. K. Khersonskiĭ, *Quantum Theory of Angular Momentum: Irreducible Tensors, Spherical Harmonics, Vector Coupling Coefficients, 3nj Symbols*, World Scientific, Teaneck, NJ, 1988.

[21] L. Wei, *New formula for 9-j symbols and their direct calculation*, Comput. Phys., 12 (1998), pp. 632–634.

[22] L. Wei, *Unified approach for exact calculation of angular momentum coupling and recoupling coefficients*, Comput. Phys. Comm., 120 (1999), pp. 222–230.