Fast and accurate evaluation of Wigner 3j, 6j, and 9j symbols using prime factorisation and multi-word integer arithmetic

H. T. Johansson¹ and C. Forssén¹,²,³

¹Department of Fundamental Physics, Chalmers University of Technology, SE-412 96 Göteborg, Sweden
²Department of Physics and Astronomy, University of Tennessee, Knoxville, TN 37996, USA
³Physics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA

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 multi-word 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 only incur rounding errors in the least significant bits. Time spent to evaluate large multi-word integers is in turn reduced by using explicit prime factorisation 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.

I. 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 [1, 2]. 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| ≤ j. Accurate evaluation of symbols with large j using floating-point arithmetics 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 multi-word integer representations in computer codes. An earlier work by L. Wei [3] demonstrated a clever reduction of the huge integers by expressing the terms as products of binomial coefficients. However, high-accuracy approaches are in general significantly slower than corresponding floating-point implementations. The published code of Ref. [3] is no exception. We find, e.g., that the computation of 6j symbols has already an order of magnitude slower than the popular GNU Scientific Library (GSL) [4] for small angular momenta, max(j) = 3, and the difference is growing 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 factorised 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.

II. METHOD

The implementation uses the fact that the xj symbols can be written on the form

$$W_x = \frac{n\sqrt{x}}{q},$$  \hspace{1cm} (1)

where n, s and q are integers. This general expression comes directly from the defining equations [1]. E.g., for 6j symbols we have the Racah formula

$$\left\{ \begin{array}{ccc} a & b & e \\ d & c & f \end{array} \right\} = \prod_{i=1}^{4} \sqrt{\Delta(\hat{a}_i)} \sum_{k} (-1)^k \frac{(k+1)!}{\prod_{j=1}^{4} (k - \alpha_j)! \prod_{j=1}^{3} (\beta_j - k)!},$$  \hspace{1cm} (2)

with

$$\Delta(a,b,c) = \frac{(a+b-c)!(a-b+c)!(b-c+a)!}{(a+b+c+1)!},$$  \hspace{1cm} (3)

and

$$\hat{a}_1 = (a,b,c), \quad \alpha_1 = a + b + c, \quad \beta_1 = a + b + c + d,$$
$$\hat{a}_2 = (d,c,e), \quad \alpha_2 = d + c + e, \quad \beta_2 = a + d + e + f,$$
$$\hat{a}_3 = (a,c,f), \quad \alpha_3 = a + c + f, \quad \beta_3 = b + c + e + f,$$
$$\hat{a}_4 = (d,b,f), \quad \alpha_4 = d + b + f.$$  \hspace{1cm} (4)

The summation index k is in the range max(\alpha_1, \alpha_2, \alpha_3, \alpha_4) ≤ k ≤ 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 a symbol not to be trivially zero, the sum \alpha_i must be integer and \hat{a}_i must fulfill triangle conditions, e.g. for \hat{a}_1:

$$|a-b| \leq e \leq a+b.$$  \hspace{1cm} (5)

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) can be brought to the form (1).

Finding the LCD is straightforward by expressing each factorial as an explicit product of prime numbers, e.g. 7! = 2 \times 3 \times 5 \times 7, treated as the tuple (4, 2, 1, 1) in the calculations. The denominator product for each term is found by adding the exponents for each prime number, and the LCD is then found by taking the maximum value for each exponent. The exponents for the nominator in each term are also included with opposite sign, which in practice reduces the LCD considerably. Actually, the LCD is kept with exponents being both positive and negative, thus also representing factors common to all terms. The sum is then evaluated by converting the now pure nominator remaining exponent tuples of each term into exact integers using multi-word arithmetics, which can then be accumulated, yielding one part of n.

The product of the \( \Delta \)-factors is formed by summing the factorial prime exponent contributions. 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 Eq. (1) while the remaining ones will be divided by two, thereby evaluating the square root, and added to the inverse LCD. These final exponents are used to create q and the second part of n, by using the negative and positive exponents, respectively.

In the final step, the value of \( W_2 \) in (1) can be evaluated using floating-point arithmetics with a limited loss in relative precision. The three conversions from integer to floating point, together with one square root, one multiplication, and one division constitute six operations that each incur the loss of (at most) half a least significant bit in the floating-point representation used, commonly referred to as the machine epsilon, \( \epsilon \). For the 64-bit floating-point format of IEEE 758, which is usually used for the C type \texttt{double}, one has \( \epsilon = 1.11 \times 10^{-16} \).

### A. Wigner 9j

Wigner 9j symbols can be written as a sum of products of 6j symbols, see e.g. Ref. [1]. It is not obvious from this direct formula for Wigner 9j symbols that they can be written on the form (1) due to the square roots in the \( \Delta \) prefactors of the 6j symbols. Nevertheless, a rearranged formula based on binomials was presented [3] and used [3] by L. Wei. This formula is used also in the present implementation, after expressing the binomials as factorials. The routine implementing the sum in the 6j formula is re-used, and in total the 9j evaluation becomes a double sum, where an overall LCD expressed on prime-exponent form can be extracted.

### III. RESULTS

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

Results are presented in Fig. 1 and Fig. 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 non-trivially-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 \texttt{369j}. The relative error never exceeded \( 6 \epsilon \approx 6.66 \times 10^{-16} \), which is our upper bound. The slightly larger accuracy loss in \texttt{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 \texttt{wigxjpf} routines presented in this work are also very efficient. In comparison to GSL (version 1.16), \texttt{wigxjpf} is actually faster for 3j symbols up to \( \text{max}(j) = 60 \), while the difference in execution speed always remains smaller than a factor 4 up to \( \text{max}(j) = 20 \) (30) for 6j (9j) symbols. At large j the floating-point routines obviously suffer from an increasing loss of accuracy, thus making \texttt{wigxjpf} superior.

Explicit numerical values, execution times, and memory usage for a selected set of symbols are presented in Table 1. Note that we present results for extreme cases such as \( j = 50, 000 \) and \( j = 2, 000 \) for 6j and 9j symbols, respectively.

### IV. IMPLEMENTATION

Before the calculation of individual symbols, a table with precalculated factorisations of factorials is prepared. Table 1 shows the maximum factorial that can be used 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
FIG. 1. Upper panel: average evaluation time for $3j$ symbols using four different implementations: \texttt{wigxjpf} (this work, circle markers), $369j$ \cite{3} (crosses), GSL-1.16 \cite{4} (diamonds), and a simple floating-point computation (plus markers). The solid lines correspond to the exhaustive measurements where all non-trivially-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 \texttt{long double} on x86 hardware. All tests have been performed using a Xeon E3-1240v3, single-threaded @ 3.8 GHz. The dashed line indicates the expected maximum relative error of our implementation, which is six times the machine epsilon.

sum (inner sum for $9j$ symbols), for each of which temporary results (nominator-denominator exponent tuples) are stored during execution.

First, all prime numbers up to the largest factorial argument, $p$, are determined by the sieve of Eratosthenes. By simple enumeration in the basis of prime numbers, the factorisation of all integers up to $p$ are determined. Each factorisation is stored as an array of integers. The factorisation of factorials are constructed by cumulating the previous factorisations. Thus the factorisations are done without any ‘test’ divisions, i.e., checks for remainder zero.

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

The multi-word 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 which perform carry between the words for addition, subtraction and multiplication are implemented. 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
multi-word 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 non-zero bits remain, the first number is squared, starting as the prime number for the first bit. For each bit set to one, the cumulative product is multiplied by the hitherto squared result. These intermediate results are treated as multi-word integers when necessary.

Conversion from multi-word 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.

Symbols that are trivially zero can be identified using a single if-clause in combination with two’s-complement arithmetics and bit manipulation. Only symbols passing this test are evaluated. Repeated similar calculations 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.

For the comparisons presented in Sec. III we have also implemented simple floating-point routines. They use precalculated floating-point factorial values 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 multi-word integers. The accuracy loss is completely dominated by cancellation in the sums of large alternating terms.

**A. Memory usage**

The largest memory requirement of \texttt{wigxjpf} is associated with the tables of precalculated factorisations of numbers and factorials, see Table \( \text{II} \). The next largest use is the temporary arrays of prime factors of each term in the sum. Other variables that are used during computations, e.g. multi-word integers, are comparatively small in size.

**V. CONCLUSIONS AND OUTLOOK**

In conclusion, we have successfully implemented a computational method for fast and accurate evaluation of Wigner 3\( j \), 6\( j \), and 9\( j \) symbols using prime factorisation and multi-word integer arithmetics. The efficiency and accuracy of our approach has been validated via bench-
TABLE I. Values and execution times for selected $xj$ symbols. These times include setup of the precalculated tables of factorised 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 @ 3.8 GHz.

| Symbol | Value | Time  | Memory  |
|--------|-------|-------|---------|
| $3j$   | (-0.01908157979919155, 0.07 ms, 58 kB) |
| $6j$   | (-0.01265208072315355, 0.06 ms, 7.0 kB) |
| $9j$   | (0.0002812983019125448, 0.08 ms, 20 kB) |

TABLE II. Maximum factorial argument and number of terms given the maximum $j$ in a symbol. The limits can be rounded down when non-integer. These limits must be considered for temporary memory allocation. The memory requirement for the precalculated factorial tables are given in the last columns for three different values of max($j$).

| Symbol | Max factorial, $p!$ | Max terms $j$ = 100 | max terms $j$ = 1,000 | max terms $j$ = 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               |

mark calculations and comparison with existing floating-point routines and with the $369j$ code by Wei [3]. The latter algorithm is similar to our approach in the sense that it also uses prime-number factorisation and multiword integer arithmetics. However, it employs recursive calculation of binomial coefficients while our implementation, with the precalculation of a table with prime-number factorisations, offers a code that is significantly faster: comparable in speed even to the floating-point routines. In addition, the precomputed factorisation 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 work, $wigxjpf$, is available for download [6].

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Research Council under the
European Community’s Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no. 240603, and the Swedish Foundation for International Cooperation in Research and Higher Education (STINT, IG2012-5158). This material is based upon work supported in parts by the U.S. Department of Energy under Contract No. DE-AC05-00OR22725 (Oak Ridge National Laboratory). The authors would like to acknowledge valuable comments on the manuscript and the computer code by B. D. Carlsson and K. Wendt.

[1] D. Varshalovich, A. Moskalev, and V. Khersonskii, Quantum Theory of Angular Momentum: Irreducible Tensors, Spherical Harmonics, Vector Coupling Coefficients, 3nj Symbols (World Scientific Pub., 1988).
[2] D. Brink and G. Satchler, Angular Momentum, Oxford science publications (Clarendon Press, 1993).
[3] L. Wei, Comput. Phys. Commun. 120, 222 (1999).
[4] M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, P. Alken, M. Booth, F. Rossi, and R. Ulerich, GNU Scientific Library Reference Manual, 3rd ed. (See also http://www.gnu.org/software/gsl, 2013).
[5] L. Wei, Comput. Phys. 12, 632 (1998).
[6] wigxjpf computer code,” http://fy.chalmers.se/subatom/wigxjpf