New efficient programs to calculate general recoupling coefficients
Part II : Evaluation of a summation formula

V. Fack, S.N. Pitre, J. Van der Jeugt*

Department of Applied Mathematics and Computer Science,

Universiteit Gent, Krijgslaan 281–S9, B–9000 Gent, Belgium

E-mail : Veerle.Fack@rug.ac.be

Published in : Comput. Phys. Commun. **86** (1995) 105–122

Abstract

In [1] we described a program `NJFORMULA` to generate a summation formula in terms of 6-j coefficients for a general angular momentum recoupling coefficient. Here we present programs for the second part of the problem, i.e. the numerical computation of a general recoupling coefficient by evaluating the formula generated by the program `NJFORMULA`. We describe a sequential version `NJSUMMATION`, written in C, which runs on a range of computers, as well as a parallel version `NJSUMPAR`, also written in (Parallel) C, which runs on an arbitrary network of transputers.

*Senior Research Associate N.F.W.O. (National Fund for Scientific Research, Belgium)

PROGRAM SUMMARY

Title of program : `NJSUMMATION`, `NJSUMPAR`

Catalogue number :

Program obtainable from : CPC Program Library, Queen's University of Belfast,
N. Ireland (see application form in this issue).

Computers used (Operating systems) : 386/486-based PCs (MS-DOS, Linux [17]), Sun Sparc (Unix), T800 transputer system (MS-DOS).

Programming language used : C.
(Compilers : Turbo C++ [18], GNU CC [19], SPARCompiler C, 3L Parallel C [20])

No. of lines in sequential program (module + 2 example programs) : 450

No. of lines in parallel program : 500

Keywords : atomic structure, nuclear structure, scattering, general recoupling coefficient, angular momentum, Racah coefficient, $3n$-j coefficient, recursive techniques, arbitrary nested loops, parallel processing, transputer, farming technique.

Nature of physical problem : A general recoupling coefficient for an arbitrary number of (integer or half-integer) angular momenta can be expressed as a formula consisting of products of 6-j coefficients summed over a certain number of variables. Such a formula can be generated using the program `NJFORMULA` [1]. The present programs perform the evaluation of a generated formula for given data of angular momenta.

Method of solution : A summation formula for a general recoupling coefficient can contain an arbitrary number of summation variables, which is determined by the binary trees involved in the search process when generating the formula [1]. To develop a general algorithm for the evaluation of such a formula, we use recursive programming techniques which allow to implement arbitrary nested loops.

Apart from the sequential implementation `NJSUMMATION` of this algorithm, we also present a parallel version `NJSUMPAR`, which can be executed on a network of transputers. The latter uses a farming technique, where a master processor distributes work packets to several worker processors who calculate products of 6-j coefficients.

Typical running time : For the sequential program the running time is determined mainly by the number of 6-j coefficients computed in the evaluation process. Typical examples, involving upto ten thousand 6-j coefficients, take less than 0.25 seconds on the Sun Sparc, less than 0.5 seconds on the 486-based PC, and less than 50 seconds on a single T800 transputer. More complicated examples, involving upto one million 6-j coefficients, take less than 20 seconds on the Sun Sparc and less than 40 seconds on the 486-based PC.

For the parallel program we focus our attention on the advantage factor of the parallel version compared with the sequential version. On a network of 4 transputers, the parallel program is 3 to 4 times faster than the corresponding sequential version.

LONG WRITE-UP

1 Introduction

The need for angular momentum coupling coefficients arises in atomic and nuclear structure or scattering calculations. The problem of the calculation of a general recoupling coefficient for an arbitrary number of integer and half integer angular momenta has been studied by several authors [2, 3, 4] and is usually treated in two parts.

In [1] we presented a program `NJFORMULA`, which is a new approach to the first part of the problem, i.e. expressing the recoupling coefficient as a (multiple) sum over products of Racah coefficients multiplied by phase factors and square root factors. The algorithm described in [1] makes use of the method of binary tree transformations, which was first introduced by Burke [2] in the program `NJSYM`. Another approach is followed by Bar-Shalom and Klapisch [3] (and later also by Lima [4]), who calculate angular momentum recoupling coefficients by means of more advanced graphical methods as developed by Yutsis, Levinson and Vanagas [5] and explained in many books [6, 7, 8, 9]. This has given rise to a fast program `NJGRAF` [3], yielding efficient summation formulae (i.e. with a minimal number of summation variables) in most cases. However in [1] we showed that the simpler method of binary tree transformations can also be used to develop a fast program generating efficient formulae.

Both programs `NJSYM` and `NJGRAF` also provide a subroutine `GENSUM`, for the second part of the problem, i.e. the numerical evaluation of a general recoupling coefficient for which a summation formula has been generated. The program `NJGRAF` implements a modified version of `GENSUM`, which is more efficient than the version in `NJSYM`. As remarked by Bar-Shalom and Klapisch, in the original `GENSUM` of Burke, the limits of the values of the summation variables are not calculated accurately, but tests are performed on all the arguments of the 6-j coefficients in order to check whether the latter should be computed or not. In their version of `GENSUM`, Bar-Shalom and Klapisch build a matrix expressing the connections between variables and perform a sort of diagonalisation in order to find the most stringent limitations to each variable. These modifications are responsible for a noticeable part of the total gain of time for `NJGRAF` over `NJSYM`, which is upto 2 orders of magnitude.

In the present paper we describe our approach to this second part of the problem. The procedure `evaluate_formula` is the equivalent of `GENSUM` in the programs `NJSYM` and `NJGRAF`, and uses a completely different method viz recursion to calculate the sum over several variables. A general algorithm for the evaluation of a summation formula with an arbitrary number of summation variables is developed, in which nested loops of arbitrary depth are 'simulated' by means of recursive programming techniques. The fact that each step in this recursive algorithm basically handles only one summation variable, also allows to formulate a clear algorithm for the determination of the range of a summation variable.

We also discuss the possibilities for parallelisation of the numerical evaluation of a summation formula for a general recoupling coefficient. The use of parallel computers for numerical computation is of recent origin and it is conceivable that parallel systems will become easily accessible for scientists in various areas, thereby motivating this aspect of our program. The parallel implementation discussed here runs on a network of transputers, with an arbitrary configuration. A farming technique is used, with a master processor distributing work (consisting of the calculation of products of 6-j coefficients) to worker processors. It is a rather straightforward extension of a parallel program we developed for the computation of 9-j coefficients [10].

The structure of the present paper is as follows. In section 2 we discuss how a recursive algorithm can be built for the evaluation of a general summation formula, as well as how we determine the range of a summation variable using the triangle conditions. Section 3 describes the (sequential) program `NJSUMMATION` and some implementation details concerning the datastructures used and the evaluation algorithm. The parallel implementation `NJSUMPAR` of the program is described in section 4, where we show how the task can be broken up into subtasks, using the farming technique, and discuss some specific problems arising in this case. Finally in section 5 some results obtained by our programs are discussed and compared with results obtained by the program `NJGRAF` [3].

2 Method of solution

2.1 Evaluation of a general summation formula

The evaluation of an n-fold summation of the form

$$S = \sum_{k_1}\sum_{k_2}\ldots\sum_{k_n} P_{k_1,\ldots,k_n}, \tag{1}$$

where $P_{k_1,\ldots,k_n}$ stands for a product of 6-j coefficients (including phase factors and square root factors) in terms of the summation variables $k_1,\ldots,k_n$ (and of angular momenta j_i), consists basically of an algorithm with n nested for-loops :

```
determine $k_1^{(\min)}$ and $k_1^{(\max)}$
for $k_1$ from $k_1^{(\min)}$ to $k_1^{(\max)}$ do
    determine $k_2^{(\min)}$ and $k_2^{(\max)}$
    for $k_2$ from $k_2^{(\min)}$ to $k_2^{(\max)}$ do
        ...
            determine $k_n^{(\min)}$ and $k_n^{(\max)}$
            for $k_n$ from $k_n^{(\min)}$ to $k_n^{(\max)}$ do
                compute the next term $P_{k_1,\ldots,k_n}$ of the summation
                add this to the temporary result for $S$
```

For a constant value of n it is easy to write a procedure evaluating an n-fold summation. However it would be unwise to write a separate evaluation procedure for every possible value of n, all of which will be different but also very alike. The obvious means is to write only one general evaluation procedure, that takes a variable n as a parameter, for which the actual value will be given at runtime.

This can be done by making use of recursive programming techniques, which allow to 'simulate' nested for-loops of arbitrary depth [11]. To that purpose, note that the actions in every for-loop in the above algorithm are very similar : for every summation variable k_i we have to determine the range $k_i^{(\min)} \to k_i^{(\max)}$ and then perform a for-loop for k_i over this range, in which each step requires similar actions for k_{i+1}. This idea is expressed in the following recursive algorithm `UNOPT`, which describes how to evaluate a summation S_i over the variables $k_i, \ldots, k_n$, with the current values for $k_1, \ldots, k_{i-1}$, for $1 \leq i \leq n$:

```
determine $k_i^{(\min)}$ and $k_i^{(\max)}$
for $k_i$ from $k_i^{(\min)}$ to $k_i^{(\max)}$ do
    evaluate $S_{i+1}$ over $k_{i+1}, \ldots, k_n$, with current $k_1, \ldots, k_{i-1}$ and $k_i$ (recursively)
    add this $S_{i+1}$ (for current $k_i$) to the temporary result for $S_i$
```

The terminating case of the recursion is reached when $i = n + 1$; at that point the actual computation of the product $P_{k_1,\ldots,k_n}$ of the 6-j coefficients for the current values of the summation variables is performed. The complete sum S is obtained by calling this algorithm to evaluate the summation S_1 over the variables $k_1, \ldots, k_n$.

Using the Horner scheme of polynomial evaluation [12], the above algorithm can be optimised by 'pulling out' independent factors for every summation. Denoting by P_{k_i} the product of 6-j coefficients (and the corresponding phase factors and square root factors) that contain only the summation variable k_i (together with some of the summation variables $k_1, \ldots, k_{i-1}$ and some of the angular momenta), then the summation (1) can be written as :

$$\sum_{k_1} \sum_{k_2} \cdots \sum_{k_n} P_{k_1,\ldots,k_n} \quad = \quad \sum_{k_1} P_{k_1} \sum_{k_2} P_{k_2} \cdots \sum_{k_n} P_{k_n} . \qquad (2)$$

Taking this into account, we obtain the following algorithm `OPT`, to evaluate a summation S_i over the variables $k_i, \ldots, k_n$, with current values for $k_1, \ldots, k_{i-1}$, for $1 \leq i \leq n$:

```
determine $k_i^{(\min)}$ and $k_i^{(\max)}$
for $k_i$ from $k_i^{(\min)}$ to $k_i^{(\max)}$ do
    compute $P_{k_i}$
    evaluate $S_{i+1}$ over $k_{i+1}, \ldots, k_n$, for current $k_1, \ldots, k_{i-1}$ and $k_i$ (recursively)
    add $P_{k_i} \times S_{i+1}$ (for current $k_i$) to the temporary result for $S_i$.
```

The terminating case of the recursion, when $i = n + 1$, needs no computations at all and simply returns 1 as the result for S_{n+1}.

2.2 Determining the range of a summation variable k_i

As described by Burke [2] and Bar-Shalom and Klapisch [3], the range of values taken by the summation variables can be decided by examining the triangular conditions placed on the different values in the 6-j coefficients appearing in the product of the summation. Since the recursive algorithm, described in the previous section, basically considers only one summation variable k_i in each algorithm step, we can develop a straightforward algorithm for the determination of the range of this variable.

A 6-j coefficient of the following general form

$$\left\{ \begin{matrix} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{matrix} \right\} \tag{3}$$

is trivially zero unless the following four triangular conditions [13] are satisfied :

$$\triangle(j_1 j_2 j_3), \triangle(j_1 l_2 l_3), \triangle(l_1 j_2 l_3), \triangle(l_1 l_2 j_3). \tag{4}$$

A triangular condition $\triangle(t_1 t_2 t_3)$ means that

$$\triangle(t_1 t_2 t_3) \quad = \quad \left[\frac{(t_1 + t_2 - t_3)!(t_1 - t_2 + t_3)!(-t_1 + t_2 + t_3)!}{(t_1 + t_2 + t_3 + 1)!} \right]^{\frac{1}{2}} \tag{5}$$

should be non-zero or the following inequality should be satisfied :

$$|t_1 - t_3| \leq t_2 \leq t_1 + t_3. \tag{6}$$

For each 6-j coefficient appearing in the products of the summation, these triangular inequalities should be checked before actually computing the 6-j coefficient, to avoid unnecessary computations. However, some of the 6-j coefficients will also involve one or more summation variables. In that case the triangular inequalities can be used to determine the range of the summation variables, in the following way.

At each step i in the recursive algorithm, described above, we are left with determining the range of the summation variable k_i. At this point we know, from the way in which the summation is evaluated, that the preceding summation variables $k_1, \ldots, k_{i-1}$ have already been given a current numerical value. To find $k_i^{(\min)}$ and $k_i^{(\max)}$ we have to consider all possible triangular inequalities appearing in the 6-j coefficients involving k_i, which will be of the form

$$|t - t'| \leq k_i \leq t + t',$$

where t and t' stand for some j_x coefficient or k_x variable. Conditions involving also some of the summation variables $k_{i+1}, \ldots, k_n$ as a value for t or t', can be ignored at this point, since they cannot give any more information about the range of k_i. But for conditions involving only j_x angular momentum values or $k_1, \ldots, k_{i-1}$ summation variables, the values of t and t' will be specific numerical values, which does give some information about the range of k_i. In particular, the value of $k_i^{(\min)}$ will be the maximum of all appropriate values of $|t - t'|$, while the value of $k_i^{(\max)}$ will be the minimum of all appropriate values of $t + t'$.

Example

As an example we show how to obtain the range of the summation variables k_1 and k_2 in the following summation formula :

$$\sum_{k_1}\sum_{k_2}(-1)^{k_1+k_2+j_2+j_8}(2k_1+1)(2k_2+1)\left\{\begin{array}{ccc} j_1 & j_2 & j_6 \\ j_8 & j_9 & k_1 \end{array}\right\}\left\{\begin{array}{ccc} j_7 & j_3 & j_8 \\ j_4 & k_1 & j_2 \end{array}\right\}$$
$$\times\left\{\begin{array}{ccc} j_7 & j_4 & k_1 \\ k_2 & j_1 & j_9 \end{array}\right\}\left\{\begin{array}{ccc} j_5 & j_4 & j_7 \\ k_2 & j_9 & j_3 \end{array}\right\}\left\{\begin{array}{ccc} j_7 & j_1 & k_2 \\ j_4 & j_2 & j_8 \end{array}\right\}. \tag{7}$$

In the first recursive algorithm step we have to determine the range of the summation variable k_1. For the first 6-j coefficient in the formula the following four triangular conditions should be satisfied :

$$\triangle(j_1j_2j_6), \triangle(j_1j_9k_1), \triangle(j_8j_2k_1), \triangle(j_8j_9j_6),$$

only two of which involve k_1. So we already have two triangular inequalities which help to decide the range of k_1 :

$$|j_1-j_9| \leq k_1 \leq j_1+j_9 \qquad \text{and} \qquad |j_8-j_2| \leq k_1 \leq j_8+j_2.$$

In a similar way we obtain two more triangular inequalities from the second 6-j coefficient in the formula :

$$|j_7-j_2| \leq k_1 \leq j_7+j_2 \qquad \text{and} \qquad |j_4-j_8| \leq k_1 \leq j_4+j_8.$$

The third 6-j coefficient in the formula gives rise to the following triangular inequalities :

$$\triangle(j_7j_4k_1), \triangle(j_7j_1j_9), \triangle(k_2j_4j_9), \triangle(k_2j_1k_1),$$

of which we only have to consider those involving k_1 and can ignore those involving k_2 at this point. This gives one more condition to decide on the range of k_1 :

$$|j_7-j_4| \leq k_1 \leq j_7+j_4.$$

The other two 6-j coefficients in the formula involve only k_2 and will give no further information in determining the range of k_1. So finally the values of $k_1^{(\min)}$ and $k_1^{(\max)}$ are decided :

$$\begin{aligned} k_1^{(\min)} &= \max(|j_1 - j_9|, |j_8 - j_2|, |j_7 - j_2|, |j_4 - j_8|, |j_7 - j_4|), \\ k_1^{(\max)} &= \min(j_1 + j_9, j_8 + j_2, j_7 + j_2, j_4 + j_8, j_7 + j_4). \end{aligned}$$

For every value of k_1 in this range we have to compute the inner summation over k_2, for which we also have to determine the range first. The triangular inequalities involved arise first of all from the third 6-j coefficient in the formula :

$$|j_4 - j_9| \leq k_2 \leq j_4 + j_9 \qquad \text{and} \qquad |j_1 - k_1| \leq k_2 \leq j_1 + k_1.$$

Note that k_1 has a specific numerical value at this point, and so it can be used to determine the range of k_2. Also from the last two 6-j coefficients in the formula some triangular conditions can be obtained, finally giving rise to the following values for $k_2^{(\min)}$ and $k_2^{(\max)}$:

$$\begin{aligned} k_2^{(\min)} &= \max\,(|j_4 - j_9|, |j_1 - k_1|, |j_4 - j_3|, |j_9 - j_7|, |j_7 - j_1|, |j_4 - j_2|), \\ k_2^{(\max)} &= \min\,(j_4 + j_9, j_1 + k_1, j_4 + j_3, j_9 + j_7, j_7 + j_1, j_4 + j_2). \end{aligned}$$

2.3 Calculation of 6-j coefficients

After each variable of summation has been replaced by its current value, we have an expression consisting of a product of 6-j coefficients which can be computed numerically.

For the computation of a 6-j coefficient we use the classical single sum expression due to Racah [14] :

$$\begin{aligned} \left\{ \begin{array}{ccc} j_1 & j_2 & j_3 \\ l_1 & l_2 & l_3 \end{array} \right\} &= (-1)^{j_1+j_2+l_1+l_2} \Delta(j_1 j_2 j_3)\Delta(l_1 l_2 j_3)\Delta(l_1 j_2 l_3)\Delta(j_1 l_2 l_3) \\ &\times \sum_{k=k^{(\min)}}^{k^{(\max)}} \frac{(-1)^k (j_1 + j_2 + l_1 + l_2 + 1 - k)!}{k!(k-\alpha_1)!(k-\alpha_2)!(\beta_1 - k)!(\beta_2 - k)!(\beta_3 - k)!(\beta_4 - k)!}, \quad (8) \end{aligned}$$

with

$$\alpha_1 = j_1 + l_1 - j_3 - l_3\,, \qquad \alpha_2 = j_2 + l_2 - j_3 - l_3\,,$$

$$\beta_1 = j_1{+}j_2{-}j_3\,, \qquad \beta_2 = l_1{+}l_2{-}j_3\,, \qquad \beta_3 = j_1{+}l_2{-}l_3\,, \qquad \beta_4 = l_1{+}j_2{-}l_3\,,$$

and

$$k^{(\min)} = \max(\alpha_1, \alpha_2, 0)\,, \qquad k^{(\max)} = \min(\beta_1, \beta_2, \beta_3, \beta_4)\,.$$

When implementing the computation of 6-j coefficients, a look-up table for the factorials is provided, as suggested by Scott and Hibbert [15]. This is useful since factorials are often needed more than once in the computations. As described by Rao *et al.* [16], the logarithms of the factorials are stored instead of the actual values themselves, for the sake of accuracy.

3 Program description

The program `NJSUMMATION` presented here, is a module to be used together with the program `NJFORMULA`, described in an earlier paper [1]. While the program `NJFORMULA` provides routines for the *generation* of a summation formula for a general recoupling coefficient, the present program `NJSUMMATION` implements a function `evaluate_formula` for the *evaluation* of such a summation formula for given values of the angular momenta.

The module also contains some auxiliary routines, such as a procedure `read_jvals` to read actual values for the angular momenta, a function `trian` to check triangle conditions imposed on the angular momenta, a procedure `order` to order the 6-j coefficients in a formula w.r.t. the summation variables, and a procedure `comp_fac` to build a look-up table for the factorials needed in the computation of 6-j coefficients, for which a routine `comp_6j` is supplied.

In this section we describe the implementation as well as some special features concerning the functionality of these routines. We also show how to use them in a simple main program and how to combine the modules `NJFORMULA` and `NJSUMMATION`.

3.1 Datastructures

The datastructure used in this program to represent a summation formula is essentially the same as in the program `NJFORMULA`. Again the index i of a j_i coefficient is used to identify the j_i coefficient, while a negative index $-i$ is used to identify a summation variable k_i of the summation formula.

For the actual values of the angular momenta, which are integer or half-integer values, the double value is stored, which allows to work with integers. (We would like to remark that the procedure `read_jvals` expects a value $2j_i + 1$ to be given as input, this is done for reasons of compatibility with the program `NJGRAF` [3].)

A summation formula is represented by a record structure `FORMULA`, which contains first of all the same fields as the structure described in [1] : The fields `nrjs`, `nrks`, `nrsixjs` contain resp. the number of j_i coefficients in the recoupling coefficient, the number of summation variables k_i and the number of 6-j coefficients in the products of the formula. Fields `jsigns`, resp. `ksigns`, and `jsqrts`, resp. `ksqrts`, are arrays with an entry for every j_i coefficient, resp. k_i summation variable, such that

> `jsigns[i]`, resp. `ksigns[i]` = power of $(-1)^{j_i}$, resp. $(-1)^{k_i}$, in the formula,
> `jsqrts[i]`, resp. `ksqrts[i]` = power of $\sqrt{(2j_i + 1)}$, resp. $\sqrt{(2k_i + 1)}$.

The field `sixjs` is an array for which every entry is a representation of one of the 6-j coefficients of the formula, as an array with 6 elements which are j_i coefficients or k_i summation variables, represented by an index i or $-i$.

For the purposes of the present program some extra fields have been added to this structure `FORMULA` : A field `js` of the array type will be used to store the actual values for the j_i angular momenta during the evaluation of the formula. The field `asixjs` is a copy of the array `sixjs` except that it contains the actual j_i and k_i values wherever possible; e.g. if $j_1 = 2$, $j_6 = 2$, $j_8 = 4$, $j_9 = 2$, $k_1 = 4$ and `sixjs[1]`$=(1, -1, 6, 8, 9, -2)$, then `asixjs[1]`$=(2, 4, 2, 4, 2, -2)$.

In the following we will assume that the 6-j coefficients in the formula are ordered in increasing order w.r.t. the summation variables, i.e. first the 6-j coefficients with no summation variables, then the 6-j coefficients with only k_1, then the 6-j coefficients with only k_1 and k_2, and so on. This ordering will be necessary for implementing the optimised algorithm `OPT`, which 'pulls out' 6-j coefficients independent of the summation variables $k_{i+1}, \ldots, k_n$ at each step i in the algorithm. For that purpose an additional field `kp` of the array type is added to the datastructure `FORMULA`, to keep track of how many 6-j coefficients can be pulled out at different levels of the evaluation. An entry `kp[i]` will contain the number of 6-j coefficients that contain only the summation variables $k_1, \ldots, k_i$, where the entry `kp[0]` signifies the number of 6-j coefficients containing no summation variables; e.g. for the summation (7), which is already ordered according to our conventions, we will have `kp[0]` $=$ 0, `kp[1]` $= 2$ and `kp[2]` $= 5$.

3.2 Evaluating a summation formula

The function `evaluate_formula` performs the evaluation of a given summation formula, assuming that the structure of the formula as well as the actual j_i angular momentum values have been filled in properly. It returns a real (double precision) value which results from this evaluation.

Before starting the actual evaluation, it first performs some initial computations and checks. The triangle conditions are checked for all the 6-j coefficients for the triads that contain only angular momenta (by means of the function `trian`); if these are not satisfied the recoupling coefficient is trivially zero. Next it is checked if the product of 6-j coefficients is ordered as described above and if the `kp` array has already been set up properly; if necessary this ordering and building of `kp` is done here, by means of a procedure `order`. After that the array `asixjs` is initialised, as a copy of the array `sixjs`, replacing all positive indices i by the corresponding j_i value, while keeping the negative indices as they are (they correspond to summation variables k_i and will be replaced later during the recursive evaluation process).

Then the product of the 6-j coefficients containing only j_i angular momentum values, together with the corresponding square root factors, is computed. For the calculation of a 6-j coefficient we have written a function `comp_6j`, which implements formula (8) as described above (we assume a look-up table for the factorials has been built initially by the program, by means of the procedure `comp_fac`). Finally, if the formula does not contain a summation (i.e. if `nrks` $= 0$), this value (multiplied with the appropriate sign factor) is returned as the result of the function, otherwise the recursive evaluation of the summation is started by calling the function `r_eval_formula`.

The recursive function `r_eval_formula` takes as parameters the formula to be evaluated and a value indicating the step i in the algorithm, which is the index i indicating for which variable k_i the summation will be evaluated next. An additional parameter is a sign factor, which is a integer power of -1, with which the obtained sum has to be multiplied. Initially the function is called for step $i = 1$, with the sign factor being the powers of -1 containing only angular momenta. The result obtained from this initial call, multiplied with the product of the square root factors and the product of 6-j coefficients containing only angular momenta (which have been computed during the above initialisation), gives the final result for the recoupling coefficient.

In each call to `r_eval_formula` the range of a summation variable k_i is determined as described previously. Different ranges are obtained for the same summation variable depending on its position in different 6-j coefficients, and the final range is decided by keeping the maximum of all the lower limits and the minimum of all the upper limits. Next a loop is performed from this $k_i^{(\min)}$ to $k_i^{(\max)}$, which replaces every occurrence of the index $-i$ in the 6-j coefficients with the current value of k_i. Then the product of 6-j coefficients containing k_i (but no variables $k_{i+1}, \ldots, k_n$) can be computed, which means we 'pull out' those 6-j coefficients which are independent of those further summation variables. It is at this stage that the information in the array `kp` is used. Finally the rest of the summation, i.e. over the summation variables $k_{i+1}, \ldots, k_n$, for the current value of k_i, must be evaluated. This is done by a recursive call to the function `r_eval_formula`, for step $i + 1$, with the new sign factor being the old sign factor plus the current sign factor for k_i. This result is multiplied with the current square root factor for k_i and the product of 6-j coefficients just computed, and this is added to a temporary value for the current summation.

This procedure is continued recursively till all the summation variables have been replaced by their values, i.e. until a call of `r_eval_formula` for step $n+1$. The terminating case of the recursion needs no computations and simply returns the appropriate sign -1 or $+1$, depending on the sign factor.

3.3 Main program

The following simple main program can be used to calculate a general recoupling coefficient and to evaluate it for different sets of actual angular momentum values. The input required here is compatible with the input style of the program `NJGRAF` [3].

A header file `njsummat.h` must be included, since it contains a set of declarations which are needed by the program. The program starts by generating a summation formula for a general recoupling coefficient, using the procedure `read_njsym` to read a recoupling coefficient, the function `generate_formula` to generate the formula and the procedure `write_formula` to write the obtained formula. The functionality of these routines is described in detail in [1].

Then the program builds a look-up table for the factorials, using the procedure `comp_fac`. Next it performs a loop, in each step reading a set of actual angular momentum values (using the procedure `read_jvals`), evaluating the obtained formula for this set of data (using the function `evaluate_formula`), and finally writing the obtained value for the recoupling coefficient. We also keep track of the number of 6-j coefficients actually computed during the evaluation of a recoupling coefficient, for which purpose a global variable `count` is updated automatically.

As mentioned earlier, the procedure `read_jvals` expects a value $2j_i+1$ for an angular momentum j_i. Moreover, after reading the values for the angular momenta, it also reads a line with the same number of `T` or `F` characters as the number of angular momenta, but these characters are not stored. All this is done for reasons of compatibility of the input with the program `NJGRAF`.

```
#include "njsummat.h"

main ()
{
    int nrjs, next;
    NODE *bra, *ket;
    DELTAS delta;
    FORMULA *form;
    double recup;

    read_njsym (&nrjs, &bra, &ket, &delta);
    form = generate_formula (nrjs, bra, ket);
    write_formula (form);

    comp_fac ();
    do {
      read_jvals (form);
      recup = evaluate_formula (form);
      printf ("recup = %e\n", recup);
      printf ("number of 6-j's computed = %d\n", count);
      scanf ("%d", &next);
    } while (next != 0);
}
```

An example of an input file for this program is the following :

```
18  6  1
1  2  8  3  4  9  8  9 10  6  7 11  5 11 12 10 12 13
```

```
2  4 14  6 14 15  1  7 16  3 16 17 15 17 18  5 18 13
3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3  3
T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
1
5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5  5
T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
1
7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7  7
T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T  T
0
```

3.4 The case of several independent sums

For some recoupling coefficients it happens that the summation variables in the formula can be partitioned into disjoint sets, such that the summation breaks up into several products of independent summations. Such independent factors can, and should, be evaluated separately, since this will result in a smaller number of 6-j coefficients actually being computed. Bar-Shalom and Klapisch [3] handle this case by separating the obtained formula into products of sums, in between the generation and the evaluation process. So far we have not taken this possibility into account when developing our algorithms, which means that the case of several independent sums is not yet handled in the most efficient way. However, the essential ideas of our algorithms still hold and we will show here how our routines can be used to handle this case appropriately.

It is easily seen that, in our approach, such independent sums arise only when for the recoupling coefficient (given as a two coupling trees [1]) there exist some pairs of angular momenta (j_{i_1}, j_{i_2}) such that the coupled node j_{i_1} in the bra tree and the coupled node j_{i_2} in the ket tree are essentially the same, in the sense that they have the same set of leaf nodes. Indeed, in that case the transformation of the subtree j_{i_1} in the bra tree to the subtree $j_{i_2} \equiv j_{i_1}$ in the ket tree will be independent of the rest of transformations needed in the coupling trees, and the subformula obtained will be an independent factor in the final summation formula.

So from the beginning, the original problem, i.e. a recoupling coefficient resulting in a summation formula with several independent sums, can be split into several subproblems, i.e. recoupling coefficients for which the summation formula contains only one independent sum. For each of these subproblems the routines we have supplied so far can be used efficiently. The only routine which we have to add here is a procedure `split`, which splits the two coupling trees into a set of pairs of coupling trees. For this purpose it uses the information returned by the input routines in [1], which is in the form of a list of delta functions $\delta(j_{i_1}, j_{i_2})$, indicating the pairs of equivalent angular momenta j_{i_1} and j_{i_2}. The procedure `split` returns an array with the independent bra subtrees as well as an array with the independent ket subtrees, obtained from a given bra and ket tree, which have to be supplied at calltime in the zeroth element of the corresponding arrays. Also the number of independent pairs of subtrees (apart from the main pair of trees) obtained is returned.

The above main program can easily be extended in such a way that, after reading a recoupling coefficient, it calls the procedure `split` when necessary. Next it has to generate each independent summation subformula separately, and finally it has to evaluate the product of these subformulae for a range of given j_i values. To distribute the actual j_i values over all the independent subformulae, we supply a routine `copy_jvals`. Below we give the listing of this main program.

```
#include "njsummat.h"

main ()
{
    int i, nrjs, nrtrees, next;
    NODE *bra[10], *ket[10];
    DELTAS delta;
```

```
      FORMULA *form[10];
      double recup;

      read_njsym (&nrjs, &bra[0], &ket[0], &delta);
      if (delta.n > 0)
        split (delta, &nrtrees, bra, ket);
      for (i=0; i<=nrtrees; i++) {
        form[i] = generate_formula (nrjs, bra[i], ket[i]);
        write_formula (form[i]);
      }

      comp_fac ();
      do {
        read_jvals (form[0]);
        recup = evaluate_formula (form[0]);
        for (i=1; i<=nrtrees; i++) {
          copy_jvals (form[0], form[i]);
          recup *= evaluate_formula (form[i]);
        }
        printf ("recup = %e\n", recup);
        printf ("number of 6-j's computed = %d\n", count);
        scanf ("%d", &next);
      } while (next != 0);
  }
```

4 The parallel version

Apart from the sequential program NJSUMMATION described above, we also present a parallel program NJSUMPAR to calculate a general recoupling coefficient. This parallel version is developed on a network of transputers, using a so-called farming technique, which makes the program completely independent of the actual transputer configuration. The method used here is similar to the method we used earlier when developing a parallel program to calculate 9-j coefficients [10].

4.1 Transputers and parallel programming

For our purposes a transputer can be viewed as a processor that can run program code independently and that has four links, through which it can communicate with other processors in the network. Our specific system consists of 20 transputers and is hosted by four PC-AT microcomputers. For that purpose, four of the transputers have host interface capability; they are T800/20 MHz with 4Mb of RAM. The other 16 transputers are T800/20 MHz with 1Mb RAM. The PC-AT hosts run server software providing access to the PC-AT for system services such as file I/O, keyboard input and screen output. Other software (such as compilers, applications, etc.) runs on the transputer nodes. The parallel program `NJSUMPAR` is written using the stand-alone Parallel C compiler from 3L [20], which is basically a standard C compiler extended with several features supporting the concurrency offered by a transputer system.

Parallel C uses the common abstract model of parallel processing in transputer systems, which is based on the idea of *communicating sequential processes*. A complete application is viewed as a collection of concurrently active sequential processes (*tasks*), which can communicate with each other over *channels*, each channel connecting one process to one other process. It is important to note that such channel communications are synchronised : a process wishing to send a message over a channel is forced to wait until the receiving process reads the message. In that way, tasks can be treated as the atomic building blocks for parallel programs, being software 'black boxes' connected together by channels.

One possible type of application in Parallel C is a so-called *farming application*, which can run on an *arbitrary* network of transputers. A problem is suited for such an implementation when its solution consists in applying the same technique several times to different data. In the processor farm technique an application consists of one *master* task, organising the work, and any number of anonymous *worker* tasks, all performing similar parts of the work (thus all running the same code). Sending work packets by the master to the workers or returning results by the workers to the master, is done transparently by *routing software* which is supplied with the Parallel C compiler.

Another important concept, supported by Parallel C, is that of a task being multi-threaded. This means that a task can contain any number of concurrent processes (called *threads*) running on the same processor, each of which is independently executing the code of the task. Each thread has its own private stack of data but shares the rest of its data with all the other threads in the same task. Threads within a task can communicate with each other via shared memory. The software construct of semaphores, which is commonly used in order to prevent threads from interfering with each other while operating on shared data, is available. Alternatively, internal channels can be used to synchronize the threads' operations and transmit data between them by passing messages.

4.2 The master task of `NJSUMPAR`

The task `MASTER` of the farming application `NJSUMPAR` runs on the master transputer, which is a transputer with host interface capability. It is responsible for organising the work, which means first of all breaking up the problem into independent work packets which are to be processed by the workers. In this case a work packet will consist of the computation of one term in the summation, which means the computation of a product of 6-j coefficients multiplied by a constant factor.

Thus the master program has all the units of the sequential program `NJSUMMATION` except for the routines involved with the actual computation of 6-j coefficients. It contains the recursive subroutine `evaluate_formula` which does the job of finding the ranges of different summation variables and then performs nested for-loops over these ranges recursively as described before. The major difference here is that once we have a product of 6-j coefficients with all the actual values filled in, its value has to be computed by a worker task.

Another difference in the parallel version is the fact that we use the unoptimised algorithm `UNOPT` instead of the optimised algorithm `OPT`. This is done because pulling out independent products of 6-j coefficients for every summation, introduces a sort of data-dependency between the recursion steps, which makes it quite non-trivial to parallelise the optimised algorithm. However, trying to do so will be a topic of our further research, since optimising the algorithm in this way yields a sizeable reduction in the computation time (as is shown in section 5).

The master task also takes care of sending the work packets to the workers, for which it uses the Parallel C procedure `net_send`. This will send a work packet to a free worker, i.e. a worker who is ready to receive a work packet. If no workers are free, the `net_send` procedure will 'sleep' and the sending process will be blocked until a worker becomes free. Finally the master also has to collect the results from the workers, which have to be added in order to obtain the final result to the problem. For this, it uses the Parallel C procedure `net_receive`, which is also blocking, i.e. the process 'sleeps' until one of the workers is ready to send a result. Because of the fact that the `net_send` and `net_receive` procedures are blocking, the master task must be a two-threaded task, in which the operations of sending work packets to the workers and receiving results from the workers have to be performed in parallel. Otherwise the program would end in a 'deadlock' situation, when the master is waiting for a worker to become free, but no worker can return its result (and become free) because the master is not ready to receive this result.

Thus there is a send thread, which takes care of building the work packets (in a recursive algorithm as described above) and of sending these work packets to the workers (which happens at the lowest level in the recursion). There is also a receive thread performing a loop, which receives the result packets from the workers and adds them to obtain the final recoupling coefficient. An additional problem here is the fact that initially it is not known how many work packets will be sent by the master and thus how many packets the receive thread must receive. To solve this problem we can use semaphores in the following way. One semaphore is used as a sort of counter indicating how many packets the send thread has sent : it is updated by the send thread each time it sends a packet and it is consulted by the receive thread to check if there are unreceived packets. To indicate that all packets have been sent, another semaphore is set by the send thread, which is checked at the appropriate time by the receive thread. The receive thread can finish its loop when there are no unreceived packets and the send thread has sent all its packets.

4.3 The worker task of `NJSUMPAR`

The task `WORKER` runs on several slave transputers and performs most of the actual computation. It basically consists of a simple sequential loop in which it accepts work packets from the master, performs the processing for which it is programmed on the data of the received work packet and sends the result back to the master.

Since the essential work done by the `WORKER` tasks is the computation of products of 6-j coefficients, each `WORKER` task starts by building a look-up table for the factorials, which contains the logarithms of factorials (as described earlier).

Next it performs a 'loop forever', computing products of 6-j coefficients, in the following way. Using the Parallel C procedure `net_receive`, it reads from 'the network' a one-record message from the master, which contains a constant factor and a list of actual 6-j coefficients that have to be computed. It then computes the product of the received 6-j coefficients, using the procedure `comp_6j` as described above. Finally it sends the computed result, multiplied by the given constant factor, back over the network to the master task, using the Parallel C procedure `net_send`.

5 Results and Discussion

The program `NJSUMMATION` is written in C and can be run on any system providing a C (or C++) compiler. We have tested the program on a range of machines (such as 386/486-based PCs running MS-DOS and Linux [17], a Sun Sparc running Unix, and a single T800 transputer hosted by a PC running MS-DOS) using several compilers (such as Turbo C++ [18], Gnu CC [19], SPARCompiler C, and 3L Parallel C [20]). The parallel version `NJSUMPAR` is written in Parallel C and runs on an arbitrary network of T800 transputers.

To compare the results obtained by our program `NJSUMMATION` with the results obtained by the program `NJGRAF` [3], we have compiled the program `NJGRAF` on the Sun Sparc using the Sparc Fortran-77 compiler and on the transputer system using the 3L Parallel Fortran compiler [21].

The test cases for which we want to discuss our results, are a selection of those described in [1], corresponding to the following recoupling coefficients :

$$(G_4)\quad \begin{array}{l}\langle((j_1,j_2)j_{12},(j_3,(j_4,(j_5,((j_6,j_7)j_{13},(j_8,(j_9,(j_{10},j_{11})j_{14})j_{15})j_{16})j_{17})j_{18})j_{19})j_{20})j_{21}\\ |(j_5,(j_7,((j_6,(j_{11},(j_9,(j_{10},j_8)j_{22})j_{23})j_{24})j_{25},(j_1,(j_3,(j_2,j_4)j_{26})j_{27})j_{28})j_{29})j_{30})j_{21}\rangle\end{array}$$

$$(F_3)\quad \langle(((j_1,j_2)j_7,(j_3,j_4)j_8)j_9,(j_5,j_6)j_{10})j_{11}\,|((j_3,j_6)j_{12},((j_2,j_4)j_{13},(j_1,j_5)j_{14})j_{15})j_{11}\rangle$$

$$(F_4)\quad \langle(((j_1,j_2)j_7,(j_3,j_4)j_8)j_9,(j_5,j_6)j_{10})j_{11}\,|((j_1,j_6)j_{12},((j_3,j_5)j_{13},(j_2,j_4)j_{14})j_{15})j_{11}\rangle$$

$$(F_5)\quad \begin{array}{l}\langle\ (((j_1,j_2)j_8,(j_3,j_4)j_9)j_{10},((j_5,j_6)j_{11},j_7)j_{12})j_{13}\\ |\ ((j_1,j_7)j_{14},(((j_3,j_5)j_{15},(j_2,j_4)j_{16})j_{17},j_6)j_{18})j_{13}\ \rangle\end{array}$$

$$(F_6)\quad \begin{array}{l}\langle\ (((j_1,j_2)j_8,(j_3,j_4)j_9)j_{10},(j_5,(j_6,j_7)j_{11})j_{12})j_{13}\\ |\ (j_5,((j_6,(j_2,j_4)j_{14})j_{15},(j_3,(j_1,j_7)j_{16})j_{17})j_{18})j_{13}\ \rangle\end{array}$$

$$(F_7)\quad \begin{array}{l}\langle\ ((j_1,(j_2,j_3)j_8)j_9,((j_4,j_5)j_{10},(j_6,j_7)j_{11})j_{12})j_{13}\\ |\ (((j_1,j_4)j_{14},j_6),j_{15},((j_5,j_2)j_{16},(j_7,j_3)j_{17})j_{18})j_{13}\ \rangle\end{array}$$

$$(F_8)\quad \begin{array}{l}\langle\ (((j_1,j_2)j_9,j_3)j_{10},(((j_4,j_5)j_{11},j_6)j_{12},(j_7,j_8)j_{13})j_{14})j_{15}\\ |\ (((j_1,j_4)j_{16},j_7)j_{17},(((j_2,j_5)j_{18},(j_8,j_3)j_{19})j_{20},j_6)j_{21})j_{15}\ \rangle\end{array}$$

$$(F_9)\quad \begin{array}{l}\langle\ (((j_1,(j_2,j_3)j_{11})j_{12},((j_4,j_5)j_{13},j_6)j_{14})j_{15},(((j_7,j_8)j_{16},j_9)j_{17},j_{10})j_{18})j_{19}\\ |\ (((j_2,j_4)j_{20},j_7)j_{21},((((j_1,j_8)j_{22},(j_9,j_5)j_{23})j_{24},j_{10})j_{25},(j_6,j_3)j_{26})j_{27})j_{19}\ \rangle\end{array}$$

Of these recoupling coefficients, only (G_4) gives rise to a summation formula with several independent sums; this is a case with two independent sums. So in order to test the version of the program handling the case of several independent sums, we have constructed some test cases by combining a pair of simpler recoupling coefficients (as the ones given above) in a form of *'direct product'*. This is obtained by coupling the bra (resp. ket) trees of both recoupling coefficients of the pair to form the bra (resp. ket) tree of the combined recoupling coefficient. The following combinations have been made here :

$$(F_{10}) = (F_3) \times (F_4)$$

$$(F_{11}) = (F_4) \times (F_7)$$

$$(F_{12}) = (F_7) \times (F_8)$$

In table 1 we compare the results obtained by the sequential program `NJSUMMATION` with the results of the program `NJGRAF`. For each test case (F_3)–(F_9) we have calculated the value of the recoupling coefficient for several sets $(2j_i + 1)$ values, all of which are chosen to be equal. Both programs do indeed obtain the same value. Since running times (given in seconds, obtained on a Sun Sparc computer) are very small for most cases, we also compare the number of 6-j coefficients actually computed during the evaluation of the recoupling coefficient. Since none of these examples contain several independent sums, the non-splitting version of `NJSUMMATION` has been used to generate these results, but of course the splitting version yields exactly the same results.

For those cases where both programs have obtained a similar formula, i.e. with the same number of summation variables and the same number of 6-j coefficients in the product of the formula, such as (F_3) and (F_6), it is clear from the table that both programs compute the same number of 6-j coefficients, so both evaluation algorithms are equally efficient.

For test cases where `NJSUMMATION` uses a more efficient summation formula than `NJGRAF`, such as (F_4) and (F_5), table 1 confirms the idea that this will yield a more efficient evaluation of the formula, with less 6-j coefficients to be computed. Moreover we noticed that for larger j_i values the program `NJGRAF` crashes during the evaluation, giving the error message *'Not a Number'* as a result; it is not clear to us why such a crash happens in `NJGRAF`.

We also give some results for cases in which `NJGRAF` does not generate a summation formula, such as (F_7), (F_8) and (F_9). Also for these cases the program `NJSUMMATION` is able to evaluate the formula in a reasonable time, such as 25 seconds on the Sun Sparc to compute 1.8 million 6-j coefficients in a seven-fold summation (F_8).

Table 2 gives results obtained for recoupling coefficients for which independent sums arise, i.e. (G_4), (F_{10}), (F_{11}) and (F_{12}). Again for each test case both the number of 6-j coefficients computed and the running time are given; running times are now obtained on a 486-based PC running Linux. It is obvious that taking the independent sums into account results in a more efficient evaluation process, with fewer 6-j coefficients to be computed. For large summations the advantage factor is even of several orders of magnitude.

For the case (G_4) `NJGRAF` too gives results which we can compare with ours. As shown in table 2, the number of 6-j coefficients computed by `NJGRAF` is even less than by our splitting version of `NJSUMMATION`. Still it is of the same order of magnitude. The reason for this difference lies in the fact that, although both `NJGRAF` and `NJSUMMATION` obtain a summation formula with the same number of summation variables k_i, these formulae are different. And for the case of all equal j_i values the formula obtained by `NJGRAF` turns out to be evaluated somewhat more efficiently than the formula obtained by `NJSUMMATION`.

For the other examples `NJGRAF` gives no results to compare with ours. Only for the case (F_{10}), `NJGRAF` can generate a summation formula, which is less efficient than the formula generated by `NJSUMMATION`. However this formula can only be evaluated for the smallest set of j_i values; for larger values `NJGRAF` crashes again (as already described above). For the other recoupling coefficients, (F_{11}) and (F_{12}), `NJGRAF` does not generate a summation formula.

Another interesting feature of the splitting version versus the non-splitting version is the fact that the splitting version is also more efficient for the *generation* of the formula. E.g. for (F_{11}) generating the formula by the non-splitting version takes 0.31 seconds, while the splitting version takes only 0.05 seconds; for (F_{12}) we get 2.83 seconds versus 0.22 seconds (on the PC). This can easily be explained by the fact that the splitting version also reduces the generation process to the generation of several subformulae. Such a subgeneration involves a search process on a coupling subtree, which is much smaller than the complete coupling tree. The complexity of such a subsearch is several orders of magnitude smaller than the complexity of a search on the complete tree.

Table 3 shows a comparison between the parallel version `NJSUMPAR` and the sequential version `NJSUMMATION`. Since the parallel version implements the unoptimised algorithm `UNOPT`, we compare it with a sequential implementation of this unoptimised algorithm, as well as with the optimal sequential version `NJSUMMATION` which implements the algorithm `OPT`. To give an idea of the difference in efficiency between these two algorithms, we also list the number of 6-j coefficients actually computed during the evaluation, for both algorithms. For small summations, such as (F_4) and (F_5), the difference in the number of 6-j coefficients computed is about a factor of 1.5 to 2, while for larger summations, such as (F_7), (F_8) and (F_9), this is already a factor of about 3. This fact is also clear from the running times (obtained on a single transputer) for the optimised and unoptimised sequential version.

When comparing the parallel version (run on a network of 4 transputers, consisting of 1 master and 3 slaves) with the unoptimised sequential version, it is clear that we get a speed-up factor of about 3 on the average, which is a good efficiency. But since optimising the sequential version also gives a speed-up factor, the parallel version gives only a speed-up factor of 2 for smaller summations and a very small speed-up for larger summations, when compared with the optimised sequential version. Therefore we conclude that a parallel implementation using the unoptimised algorithm does not gain much more than simply optimising the sequential algorithm. The remaining question is then : can the optimised evaluation algorithm be parallelised? This problem is quite non-trivial, since pulling out products of 6-j coefficients introduces a sort of data-dependency between the different recursion steps. However, constructing techniques similar to parallel backtracking, we hope to develop an optimised parallel version resulting in a similar speed-up w.r.t. the optimised sequential version.

Acknowledgements

It is a pleasure to thank Prof. K. Srinivasa Rao (IMSC, Madras, India) for stimulating discussions. This work was partly supported by the EEC (contract no. CI1*–CT92–0101).

References

[1] V. Fack, S.N. Pitre and J. Van der Jeugt, *New efficient programs to calculate general recoupling coefficients, part I : Generation of a summation formula*, Comput. Phys. Commun. **83** (1994) 275.

[2] P.G. Burke, Comput. Phys. Commun. **1** (1970) 241.

[3] A. Bar-Shalom and M. Klapisch, Comput. Phys. Commun. **50** (1988) 375.

[4] P.M. Lima, Comput. Phys. Commun. **66** (1991) 89.

[5] A.P. Yutsis, I.B. Levinson and V.V. Vanagas, The Theory of Angular Momentum (Israel Program for Scientific Translation, Jerusalem, 1962).

[6] L.C. Biedenharn and J.D. Louck, The Racah-Wigner algebra in Quantum Theory, Encyclopedia of Mathematics and its Applications, Vol. 9, ed. G.-C. Rota (Addison Wesley, Reading, MA, 1981).

[7] D.M. Brink and G.R. Satchler, Theory of Angular Momentum (Clarendon Press, Oxford, 1968).

[8] B.R. Judd, Operator Techniques in Atomic Spectroscopy (McGraw-Hill, New York, 1963).

[9] E. El Baz and B. Castel, Graphical Methods of Spin Algebras (Marcel Dekker, New York, 1972).

[10] V. Fack, J. Van der Jeugt and K. Srinivasa Rao, Comput. Phys. Commun. **71** (1992) 285.

[11] J.S. Rohl, Recursion via Pascal (Cambridge University Press, Cambridge, 1984).

[12] K. Srinivasa Rao, Comput. Phys. Commun. **22** (1981) 227.

[13] A. Edmonds, Angular momentum in quantum mechanics (Princeton University Press, Princeton, 1957).

[14] G. Racah, Phys. Rev. **62** (1942) 438.

[15] N.S. Scott and A. Hibbert, Comput. Phys. Commun. **28** (1982) 189.

[16] K. Srinivasa Rao, V. Rajeswari and Charles B. Chiu, Comput. Phys. Commun. **56** (1989) 231.

[17] Linux (Unix clone for 386/486-based PCs) version 1.0; publicly available via anonymous ftp to `nic.funet.fi` in directory `/pub/OS/Linux`.

[18] Turbo C++ (version 1.01) User Guide, Borland International Inc. (1990).

[19] GNU CC version 2.5, Free Software Foundation, Cambridge, MA, USA; publicly available via anonymous ftp to `prep.ai.mit.edu` in directory `/pub/gnu`.

[20] Parallel C (version 2.2.2) User Guide, 3L Ltd., UK (1991).

[21] Parallel Fortran (version 2.1.3) User Guide, 3L Ltd., UK (1990).

Table 1: Comparison of results obtained by the sequential version `NJSUMMATION` and the program `NJGRAF`. For each test case we list the results obtained for several sets of all equal $(2j_i + 1)$ values, which are shown in column 2, while column 3 shows the corresponding value of the recoupling coefficient. For both programs the first column gives the number of summation variables k_i and the number of 6-j coefficients in the formula used, while the second column shows the number of 6-j coefficients actually computed during the evaluation process, and the third column contains the running time in seconds on a Sun Sparc computer.

			`NJSUMMATION`			`NJGRAF`		
	j's	value	formula	#6-j's	time	formula	#6-j's	time
(F_3)	3	6.250 000E−2	2 – 6	27	0.00	2 – 6	27	0.00
	5	−3.240 837E−2		63	0.00		63	0.00
	7	−7.638 889E−2		114	0.01		114	0.01
	9	−1.570 528E−2		180	0.01		180	0.01
	11	1.166 757E−2		261	0.01		261	0.01
(F_4)	9	−1.570 528E−2	2 – 6	222	0.01	3 – 7	1137	0.07
	11	−1.166 757E−2		326	0.02		2022	0.10
	13	4.549 656E−2		450	0.02		3280	0.13
	15	−5.691 117E−3		594	0.02		4977	0.21
	25	1.332 245E−2		1 614	0.04		–	–
(F_5)	5	−3.781 349E−2	2 – 7	68	0.01	4 – 9	894	0.04
	7	7.060 185E−2		121	0.01		3101	0.13
	9	−2.024 054E−2		189	0.02		–	–
	15	−1.487 217E−2		483	0.03		–	–
(F_6)	7	6.944 444E−3	2 – 7	139	0.01	2 – 7	139	0.01
	9	−4.415 162E−3		223	0.02		223	0.02
	11	−1.759 274E−3		327	0.02		327	0.02
	17	−1.380 111E−4		759	0.03		759	0.02
(F_7)	3	1.250 000E−1	4 – 9	150	0.02	–	–	–
	5	6.537 455E−3		868	0.03		–	–
	7	−9.259 259E−3		2 985	0.06		–	–
	9	3.567 917E−3		7 701	0.11		–	–
	11	1.213 792E−2		16 594	0.25		–	–
	31	−3.657375E−4		952 689	19.35		–	–
(F_8)	5	−2.286 994E−4	7 – 13	37 212	0.69	–	–	–
	7	2.700 617E−3		340 368	5.15		–	–
	9	−1.299 748E−3		1 846 922	25.47		–	–
(F_9)	3	3.906 250E−3	9 – 17	7 482	0.62	–	–	–
	5	−2.027 983E−3		480 752	7.07		–	–
	7	−7.019 462E−4		8 535 793	109.86		–	–

Table 2: For some recoupling coefficients in which independent sums arise, we compare results obtained by the non-splitting and the splitting version of the sequential program `NJSUMMATION`, as well as results of the program `NJGRAF`. For each test case we give results obtained for several sets of all equal $(2j_i + 1)$ values, which are shown in column 2, while column 3 lists the corresponding value of the recoupling coefficient. For both programs the first column gives the number of summation variables k_i and the number of 6-j coefficients in the formula used. For `NJSUMMATION` the second and third column list the number of 6-j coefficients computed and the running time, for the non-splitting version, while the fourth and fifth column list the same data for the splitting version. The same data is listed for `NJGRAF`. Running times are given in seconds and are obtained on a 486-based PC running Linux.

			`NJSUMMATION`					`NJGRAF`		
				non-splitting		splitting				
	j's	value	formula	#6-j's	time	#6-j's	time	formula	#6-j's	ti
(G_4)	3	2.241 607E−17	3 – 11	117	0.01	45	0.01	3 – 11	36	0.
	5	−1.125 161E−04		463	0.03	103	0.01		76	0.
	7	7.822 823E−18		1 193	0.07	185	0.01		131	0.
	9	−1.085 478E−04		2 451	0.13	291	0.02		201	0.
(F_{10})	3	−3.906 250E−03	4 – 12	192	0.01	57	0.01	5 – 13	564	0.
	5	1.050 303E−03		1 082	0.05	137	0.01		–	
	7	−5.835 262E−03		3 672	0.17	252	0.02		–	
	9	2.466 559E−04		9 402	0.44	402	0.03		–	
(F_{11})	3	−7.812 500E−03	6 – 15	930	0.08	180	0.01	–	–	
	5	−2.118 683E−04		13 962	0.57	942	0.05		–	
	7	−7.073 045E−04		92 673	4.05	3 123	0.14		–	
	9	−5.603 515E−05		392 973	17.50	7 923	0.37		–	
(F_{12})	3	−1.171 875E−02	11 – 22	49 585	2.60	1 735	0.11	–	–	
	5	−1.495 112E−06		8 080 900	380.01	38 080	1.72		–	

Table 3: Comparison of the parallel version `NJSUMPAR` with the sequential version `NJSUMMATION`. For each test case we list the results obtained for several sets of all equal $(2j_i + 1)$ values, which are shown in column 2, while column 3 shows the corresponding value of the recoupling coefficient. We compare running times for the parallel (i.e. unoptimised) version, the sequential unoptimised version and the sequential optimised version (i.e. the version discussed in table 1). Running times are given in seconds and are obtained on a network of 4 transputers for the parallel version and on a single transputer for the sequential versions. We also list the number of 6-j coefficients actually computed for both the optimised and unoptimised version.

			parallel	sequential			
			`UNOPT`	`UNOPT`		`OPT`	
	j's	value	time	#6-j's	time	#6-j's	time
(F_4)	11	−1.166 757E−2	0.074	456	0.205	326	0.155
	21	−4.668 661E−3	0.247	1 656	0.773	1 146	0.544
	31	8.127 933E−3	0.561	3 606	1.833	2 466	1.246
(F_5)	9	−2.024 054E−2	0.055	357	0.155	189	0.093
	15	−1.487 217E−2	0.138	987	0.436	483	0.230
	31	5.938 541E−3	0.553	4 207	2.124	1 927	0.996
	41	−4.149 020E−3	0.953	7 357	4.077	3 317	1.828
(F_7)	5	6.537 455E−3	0.291	1 953	0.781	868	0.397
	9	3.567 917E−3	2.535	19 719	7.711	7 701	3.412
	11	1.213 792E−2	5.458	43 749	17.300	16 594	7.386
(F_8)	3	−9.375 000E−2	0.721	4 277	1.812	1 585	0.786
	5	−2.286 994E−4	16.176	124 020	47.502	37 212	16.952
	7	2.700 617E−3	145.639	1 233 596	462.954	340 368	151.947
(F_9)	3	3.906 250E−3	3.662	20 672	9.070	7 482	3.765
	5	−2.027 983E−3	207.980	1 634 346	628.240	480 752	214.418