New efficient programs to calculate general recoupling coefficients
Part I : Generation 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. **83** (1994) 275–292

Abstract

A program to generate a summation formula in terms of 6-j coefficients for a general angular momentum recoupling coefficient is described. This algorithm makes use of binary tree transformations as introduced by Burke [1] in the program `NJSYM`. Due attention is paid to finding an optimal summation formula with a minimal number of summation variables, thereby improving the results of `NJSYM`. The results obtained here are at least as good as (and often better than) the results of the alternative approach `NJGRAF`, introduced by Bar-Shalom and Klapisch [2], using more advanced graphical methods.

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

PROGRAM SUMMARY

Title of program : `NJFORMULA`

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 [11]), Sun Sparc (Unix), T800 transputer system (MS-DOS).

Programming language used : C
(Compilers : Turbo C++ [12], GNU CC [13], SPARCompiler C, 3L Parallel C [14]).

No. of lines in source program (module + 2 example main programs) : 950

Keywords : atomic structure, nuclear structure, scattering, general recoupling coefficient, angular momentum, Racah coefficient, $3n$-j coefficient, coupling tree, binary trees, recursive search process.

Nature of physical problem : A general recoupling coefficient for an arbitrary number of (integer or half-integer) angular momenta is expressed as a multiple sum over products of 6-j coefficients, including phase factors and square root factors. This summation formula can then be evaluated for given values of the angular momenta (for this purpose we will provide a program `NJSUMMATION` [3]).

Method of solution : A summation formula for a general recoupling coefficient is obtained by determining a path between the binary coupling trees corresponding to the state vectors, using so-called flop operations. Such a path consists of a number of 'successful flop sequences', each of which is found by a recursive search process. To obtain an optimal

summation formula, i.e. with a minimal number of summation variables, the total number of flop operations must be minimal. Heuristic techniques are used to keep the length of the path as short as possible.

Typical running time : For typical examples, obtaining single, double or triple summation formulae, the running time is of the order of 0.01 second on the Sun Sparc and on a 486-based PC. For more complicated examples, such as generating a 7-fold and a 9-fold summation formula, the program takes less than 0.5 second. On the whole we find our running times to be comparable with those of `NJGRAF` [2].

LONG WRITE-UP

1 Introduction

In this paper we tackle an old problem by means of a new method. The problem is the calculation of a general recoupling coefficient for an arbitrary number of integer or half integer angular momenta. Such computations appear typically in atomic and nuclear structure or scattering calculations.

A famous program of Burke [1], `NJSYM`, is dealing with the same problem. In Burke's approach, the problem is divided in essentially two parts : first, the general recoupling coefficient is expressed as a (multiple) sum over a product of 6-j coefficients (including phase factors and square root factors); secondly, this expression is evaluated for given data of the angular momenta. Our approaches to both parts are strikingly different from the classical ones, particularly because we make extensive use of recursive programming. With this view, it is appropriate to treat the two parts of the problem as two separate topics. In the present paper we shall be dealing with the first part of the problem, namely the construction of a summation formula for the evaluation of a general recoupling coefficient. In a second paper [3], we shall present the new approaches, coming from recursive programming and parallelisation, to the remaining part of the problem, i.e. that of the actual evaluation of such a summation formula.

It follows already from the work of Burke [1] that a general recoupling coefficient can be expressed as a sum over products of 6-j coefficients. In Biedenharn and Louck [4] a special topic (Topic 12) is dealing with this problem and there this fact is formulated as a *fundamental theorem* : the transformation coefficient between each pair of binary coupling schemes is expressible in terms of sums over products of Racah coefficients (ignoring the presence of dimension and phase factors). A binary coupling scheme is simply expressing the order in which n angular momenta are being coupled,

e.g. $(n = 4)$:

$$| ((j_1, j_2)j_5, (j_3, j_4)j_6)j_7\ m \rangle = \sum_{m_1,\ldots,m_6} C^{j_1 j_2 j_5}_{m_1 m_2 m_5} C^{j_3 j_4 j_6}_{m_3 m_4 m_6} C^{j_5 j_6 j_7}_{m_5 m_6 m} | j_1 m_1 \rangle \otimes | j_2 m_2 \rangle \otimes | j_3 m_3 \rangle \otimes | j_4 m_4 \rangle \quad (1)$$

Herein, $C^{jj'j''}_{mm'm''}$ is a vector-coupling (Wigner or Clebsch-Gordan) coefficient [5, 4]. The relation between such a coupling scheme and a binary tree is clear [1, 4]. A general recoupling coefficient is then a transformation coefficient between two such coupling schemes, e.g. :

$$\langle ((j_1, j_2)j_5, (j_3, j_4)j_6)j_7 \mid (j_1, ((j_2, j_3)j_8, j_4)j_9)j_7 \rangle. \quad (2)$$

Here, the m-dependence is dropped since such coefficients are independent of m [5].

To determine a formula for the calculation of an arbitrary general recoupling coefficient (of n angular momenta), Burke [1] determined an algorithm to express this coefficient as a sum over Racah or 6-j coefficients. For example, for the recoupling coefficient (2) Burke's program `NJSYM` would yield the expression :

$$(-1)^{j_1+j_2-j_5}[(2j_5+1)(2j_6+1)(2j_8+1)(2j_9+1)]^{1/2} \times \sum_k (-1)^{j_1+j_8+k}(2k+1) \begin{Bmatrix} j_4 & j_3 & j_6 \\ j_5 & j_7 & k \end{Bmatrix} \begin{Bmatrix} j_1 & j_2 & j_5 \\ j_3 & k & j_8 \end{Bmatrix} \begin{Bmatrix} j_1 & j_8 & k \\ j_4 & j_7 & j_9 \end{Bmatrix} \quad (3)$$

This is a single sum (one summation variable k). However, this is by no means the simplest expression for (2). In fact, using for example the Biedenharn-Elliott sum rule, formula (3) becomes

$$(-1)^{j_1+2j_2+j_3+j_4+j_6+j_7+j_9} \times [(2j_5+1)(2j_6+1)(2j_8+1)(2j_9+1)]^{1/2} \begin{Bmatrix} j_1 & j_2 & j_5 \\ j_6 & j_7 & j_9 \end{Bmatrix} \begin{Bmatrix} j_2 & j_3 & j_8 \\ j_4 & j_9 & j_6 \end{Bmatrix} \quad (4)$$

From the computational point of view, it is clearly better to use formula (4) instead of (3) in order to calculate the recoupling coefficient (2), because (4) contains no sum and (3) is a single sum.

As we shall see in the following sections, (3) and (4) are both related to a path between the binary trees corresponding to the coupling schemes in the bra- and ket-vector of (2); (3) corresponds to a path with 3 steps (i.e. 3 *flops*), whereas (4) corresponds to a path with 2 steps. In order to find an optimal expression for a general recoupling coefficient, a shortest path (one with the smallest number of flops) should be found. Herein, a flop is a certain operation on a binary tree; this will be discussed in more detail in Section 2. Thus, the problem of finding an optimal expression for the calculation of a general recoupling coefficient is reduced to a graph theoretical problem : how to find the shortest path between two (labelled) binary trees using flop-operations. This turns out to be a rather difficult and intriguing problem in graph theory.

In Burke's approach [1], his method is equivalent to finding a certain path between the two binary trees. However, it is usually not a shortest path, as can be seen from the above example. In fact, the path found by means of `NJSYM` is generally rather long, thus yielding expressions which are far from optimal. In order to improve `NJSYM`, Bar-Shalom and Klapisch [2] developed a new program `NJGRAF`. Their way of producing a "best formula" (where the number and values of summation variables are minimal) was to use graphical methods as developed by Yutsis, Levinson and Vanagas [6] and explained in many books [4, 7, 8, 9]. This graphical method is very powerful, and for many transformation coefficients it will yield the most optimal expression in terms of 6-j coefficients. However, the algorithms to reduce such graphs are not easy to implement, and for a complicated graph many tests and searches need to be performed in order to find an appropriate reduction, leading to an expression in terms of sums over products of 6-j's. Nevertheless, it should be said that the program using such graphical representation, `NJGRAF`, performs quite well.

The purpose of the present paper is to point out that the method of graphical representations is not needed, and that the much simpler method of binary tree transformations can succesfully be used. One advantage is

that the algorithm is much easier and rather straightforward : there is in fact only one kind of search. The data structure is also easier : this makes the program very transparent. In order to implement such a search it is natural to use a recursive algorithm; this will be analysed and explained in the following sections. It should be mentioned that the algorithm developed here is heuristic in the sense that it produces a short path between the binary trees, however it does not necessarily produce the shortest possible path. Roughly speaking, this is because it uses consecutive local minima and does not try to determine a global minimum. To determine such a global minimum (and thus a shortest path) would be too time and space consuming and one has to find a compromise between the optimality of the generated formula and the search time and space. Nevertheless, our method is so good that for all practical cases it does produce a shortest path and thus the most optimal expression. For general recoupling coefficients that are more complicated our algorithm yields a very short path (although it is not always possible to prove that it is indeed a shortest path), and in fact for all test examples in the literature and for the examples we give ourselves, our algorithm produces an expression which is often better than (and sometimes as good as) the one obtained by means of `NJGRAF`. By a better expression, we mean a (multiple) sum over products of 6-j coefficients with fewer summation variables and fewer 6-j's. These advantages (simpler algorithm, easy data structure, better result) convinced us that our approach is worth following and in this paper we present our study in further detail. In a second paper devoted to this topic [3], we shall present a number of novelties concerning the actual calculation of a multiple sum over products of 6-j's; this will lead to a new program `NJSUMMATION` as a replacement for the classical subroutine `GENSUM` [1, 2].

The structure of the paper is as follows. In section 2 we describe how a recoupling coefficient can be represented by a pair of binary coupling trees and how a corresponding summation formula can be obtained by a sequence of basic tree operations (so-called flops and exchanges). Section 3 describes

an algorithm for constructing such a path, performing the transformation of one tree into the other; here it is also explained which ideas are used to keep the path as short as possible. In section 4 we give some implementation details about the datastructures used to represent a coupling tree and a summation formula, about the input of general recoupling coefficients, and about the transformation algorithm. Finally in section 5 some results obtained by our new approach are discussed and compared with those obtained by the program `NJGRAF` [2].

2 Method of solution

In a general recoupling coefficient, such as

$$\langle\, ((j_1, j_2)j_6, (j_3, (j_4, j_5)j_7)j_8)j_9 \mid (((j_1, j_4)j_{10}, (j_2, j_3)j_{11})j_{12}, j_5)j_9 \,\rangle, \qquad (5)$$

each binary coupling scheme or state vector can be represented by a so-called *coupling tree*, which is a binary tree, as is shown in figure 1.

Figure 1: Coupling trees for the general recoupling coefficient $\langle\, ((j_1, j_2)j_6, (j_3, (j_4, j_5)j_7)j_8)j_9 \mid (((j_1, j_4)j_{10}, (j_2, j_3)j_{11})j_{12}, j_5)j_9 \,\rangle$.

Each j_i coefficient corresponds to a *node* of the coupling tree. Some of the j_i coefficients are the coupling of two other j_i coefficients, e.g. $j_6, \ldots, j_{12}$. The corresponding node in the coupling tree has nodes down to the left and to the right, called the left and right *children* of the node. We call this a *coupled node*, to distinguish it from a *leaf node*, i.e. a node corresponding to a j_i coefficient at the lowest level of the tree, e.g. $j_1, \ldots, j_5$. Leaf nodes have no left or right children. The *root node* is the top node of a coupling tree. Note that every coupled node can be considered as the root node of a *subtree* of the coupling tree. In a recoupling coefficient the two coupling trees will have a common root node, e.g. j_9, otherwise the recoupling coefficient is trivially zero. For all practical purposes this means that the root node, and in general

every coupled node, is uniquely identified by the set of its descendent leaf nodes, an idea we will use in the algorithm described in section 3.

As described by Burke [1], a general recoupling coefficient can be expressed as a (multiple) sum over products of 6-j coefficients, including phase factors and square root factors. Such a summation formula can be obtained by a transformation process on the coupling trees corresponding to the recoupling coefficient. For that purpose a set of basic operations (*exchanges* and *flops*) on a coupling tree are defined, which correspond to simple recoupling coefficients, and thus to simple summation expressions.

An *exchange* is a transformation of a state vector of the form $|\,(a,b)c\,\rangle$ to $|\,(b,a)c\,\rangle$, for which the coupling trees are shown in figure 2, and corresponds to the following recoupling coefficient [5] :

$$\langle\,(a,b)c\,|\,(b,a)c\,\rangle \;=\; (-1)^{\pm(a+b-c)}. \tag{6}$$

Figure 2: Coupling trees for an exchange : $\langle\,(a,b)c\,|\,(b,a)c\,\rangle$.

A *flop* is a transformation of a state vector of the form $|\,((a,b)d,c)f\,\rangle$ to $|\,(a,(b,c)e)f\,\rangle$ or vice versa, for which the coupling trees are shown in figure 3, and corresponds to the following recoupling coefficient [5] :

$$\begin{aligned}\langle\,((a,b)d,c)f\,|\,(a,(b,c)e)f\,\rangle \;&\equiv\; \langle\,(a,(b,c)e)f\,|\,((a,b)d,c)f\,\rangle \\ &=\; (-1)^{\pm(a+b+c+f)}\sqrt{(2d+1)(2e+1)}\begin{Bmatrix} a & b & d \\ c & f & e \end{Bmatrix}\end{aligned} \tag{7}$$

Figure 3: Coupling trees for a flop : $\langle\,((a,b)d,c)f\,|\,(a,(b,c)e)f\,\rangle$.

In order to obtain a summation formula for a recoupling coefficient, we start from the coupling tree of the initial state vector and try to transform it

into the coupling tree of the final state vector, by applying a sequence of basic operations on subtrees of the original coupling tree. Each transformation step will work on some subtree, which matches the appropriate tree for the basic operation, in the following way : for an exchange the subtree has to be of the form $(a,b)c$, where the nodes a and b can be coupled nodes or leaf nodes of the complete coupling tree; for a flop the subtree has to be of the form $((a,b)d,c)f$ or $(a,(b,c)e)f$, where the nodes a, b and c can be coupled nodes or leaf nodes of the complete coupling tree. Each transformation step will contribute some part in the summation formula : For an exchange on a subtree $(a,b)c$ the only contribution is a sign factor of the following form :

$$(-1)^{\pm(a+b-c)}.$$

When applying a flop on a subtree $((a,b)d,c)f$, the node d is replaced by a new node e in the transformed subtree $(a,(b,c)e)f$. However, if the nodes b and c are already coupled to a node j_i in the final coupling tree, then the node $e = j_i$ is not truly new. In that case the contribution in the summation formula is only a factor, consisting of a number and a 6-j coefficient :

$$(-1)^{\pm(a+b+c+f)}\sqrt{(2d+1)(2j_i+1)}\left\{\begin{array}{ccc} a & b & d \\ c & f & j_i \end{array}\right\}.$$

In the other case, the node e is truly new, and gives rise to a new summation variable in the summation formula :

$$\sum_e (-1)^{\pm(a+b+c+f)}\sqrt{(2d+1)(2e+1)}\left\{\begin{array}{ccc} a & b & d \\ c & f & e \end{array}\right\}.$$

The complete summation formula is then obtained as the consecutive product of all the contributions of the basic operations needed to transform the initial coupling tree to the final coupling tree.

As an example we show how to obtain a summation formula for the following recoupling coefficient, which is known to be proportional to a 9-j coefficient [5] :

$$\langle\, ((j_1,j_2)j_{12},(j_3,j_4)j_{34})j \mid ((j_1,j_3)j_{13},(j_2,j_4)j_{24})j \,\rangle$$

$$= \sqrt{(2j_{12}+1)(2j_{34}+1)(2j_{13}+1)(2j_{24}+1)} \left\{ \begin{array}{ccc} j_1 & j_2 & j_{12} \\ j_3 & j_4 & j_{34} \\ j_{13} & j_{24} & j \end{array} \right\}. \quad (8)$$

Table 1 lists a sequence of flops and exchanges which performs the transformation of the coupling tree $((j_1, j_2)j_{12}, (j_3, j_4)j_{34})j$ to the coupling tree $((j_1, j_3)j_{13}, (j_2, j_4)j_{24})j$, thereby denoting the newly introduced node by k. Figure 4 shows the complete coupling tree at each stage in the transformation process.

Operation on subtree	thus creating	Contribution to summation formula
$((j_1, j_2)j_{12}, j_{34})j$	$(j_1, (j_2, j_{34})k)j$	$\sum_k (-1)^{j_1+j_2+j_{34}+j} \sqrt{(2j_{12}+1)(2k+1)} \left\{ \begin{array}{ccc} j_1 & j_2 & j_{12} \\ j_{34} & j & k \end{array} \right\}$
$(j_2, j_{34})k$	$(j_{34}, j_2)k$	$(-1)^{-j_2-j_{34}+k}$
$((j_3, j_4)j_{34}, j_2)k$	$(j_3, (j_4, j_2)j_{24})k$	$(-1)^{j_2+j_3+j_4+k} \sqrt{(2j_{34}+1)(2j_{24}+1)} \left\{ \begin{array}{ccc} j_3 & j_4 & j_{34} \\ j_2 & k & j_{24} \end{array} \right\}$
$(j_1, (j_3, j_{24})k)j$	$((j_1, j_3)j_{13}, j_{24})j$	$(-1)^{-j_1-j_3-j_{24}-j} \sqrt{(2j_{13}+1)(2k+1)} \left\{ \begin{array}{ccc} j_1 & j_3 & j_{13} \\ j_{24} & j & k \end{array} \right\}$
$(j_4, j_2)j_{24}$	$(j_2, j_4)j_{24}$	$(-1)^{-j_2-j_4+j_{24}}$

Table 1: Sequence of flops and exchanges transforming the coupling tree $((j_1, j_2)j_{12}, (j_3, j_4)j_{34})j$ to the coupling tree $((j_1, j_3)j_{13}, (j_2, j_4)j_{24})j$.

This results in the following summation formula :

$$\sqrt{(2j_{12}+1)(2j_{34}+1)(2j_{13}+1)(2j_{24}+1)} \quad (9)$$
$$\times \sum_k (-1)^{2k}(2k+1) \left\{ \begin{array}{ccc} j_1 & j_2 & j_{12} \\ j_{34} & j & k \end{array} \right\} \left\{ \begin{array}{ccc} j_3 & j_4 & j_{34} \\ j_2 & k & j_{24} \end{array} \right\} \left\{ \begin{array}{ccc} j_1 & j_3 & j_{13} \\ j_{24} & j & k \end{array} \right\},$$

yielding the well known expression for 9-j coefficients in terms of 6-j coefficients.

Figure 4: Sequence of transformations for the recoupling coefficient $\langle\, ((j_1, j_2)j_{12}, (j_3, j_4)j_{34})j \mid ((j_1, j_3)j_{13}, (j_2, j_4)j_{24})j \,\rangle$

3 Algorithm description

In this section we develop an algorithm to generate a summation formula for a given recoupling coefficient, by constructing a sequence of basic operations, as described in section 2, which transforms the initial state coupling tree into the final state coupling tree. By means of heuristic techniques, we will try to generate an *optimal* summation formula, which can be evaluated afterwards for actual j_i values in an efficient way.

An optimal summation formula will have a minimal number of summation variables. In the transformation process a new summation variable can be introduced by a flop operation, so the number of summation variables will depend on the number of flop operations in the transformation sequence. However not every flop operation introduces a new summation variable : a flop operation involving a 'new' node which is one of the (coupled) nodes of the final coupling tree, only contributes a factor containing a 6-j coefficient. This means that the number of summation variables equals the number of flop operations in the transformation process, minus the number of coupled nodes in the final state coupling tree that do not appear in the initial coupling tree (i.e. the root node). So, generating a summation formula with a minimal number of summation variables corresponds to constructing a transformation sequence for the coupling trees with a *minimal number of flop operations*.

Note that an arbitrary number of exchange operations can be allowed in the transformation sequence, since an exchange will never contribute a new summation variable.

3.1 Constructing a transformation sequence

In the following algorithm the basic idea is to try and obtain the coupled nodes of the final tree as fast as possible. This happens in a number of steps, as follows :

Each step starts from a *current* tree, which is obtained from the initial tree by a sequence of flop (and exchange) operations and which already con-

tains some of the coupled nodes of the final tree. This current tree is then transformed in such a way that one more coupled node of the final tree is obtained, in a *minimal* number of flop (and an arbitrary number of exchange) operations. How this can be done, is described below. This transformed tree is then the current tree for the next step.

Starting from the initial tree, we thus arrive at the final tree in a finite number of steps, i.e. the number of coupled nodes in the final tree not appearing in the initial tree. The number of flop operations needed for the complete transformation is the sum of the number of flop operations of all the algorithm steps. The fact that each 'local' step obtains its result in a minimal number of flops, does not *guarantee* that the final result is obtained in a *minimal* number of flops, but this 'global' minimum will be approached quite close.

In each algorithm step we search for a sequence of flop (and exchange) operations in the following way :

First we try out *all possible single flop sequences* on the current tree, to see if any one of them obtains one of the coupled nodes of the final tree. By 'single flop sequences' we mean flop operations of the form $\langle((a,b)d,c)f\,|\,(a,(b,c)e)f\rangle$ (as defined in (7)), but also operations of the form $\langle((a,b)d,c)f\,|\,(b,(a,c)e)f\rangle$, which consist of an exchange followed by a flop operation. By 'all possible single flop sequences on a tree' we mean the set of single flop sequences that can be performed on the subtrees of the tree. In the next subsection we describe how this can be obtained by traversing the coupling tree. By 'obtaining a node of the final tree' we mean obtaining a node which has the same set of leaf nodes as one of the nodes in the final tree, since a coupled node is uniquely identified by the set of its leaf nodes. We can perform any succesful single flop sequence; this gives rise to a new factor (containing one 6-j coefficient) in the summation formula, but not to a new summation variable.

If no single flop sequence obtains a coupled node of the final tree, then we try all possible double flop sequences to see if any one of them results

in a coupled node of the final tree. We can perform any successful double flop sequence, giving rise to two 6-j coefficients and one summation variable. If no double flop sequence obtains a coupled node of the final tree, we try all possible sequences with three flops, and so on until at last we obtain a coupled node of the final tree. The next subsection describes in detail how this can be performed by a recursive algorithm. For such a sequence with i flops, the contribution in the summation formula will be a product of i 6-j coefficients and $(i-1)$ summation variables.

In each step of the algorithm there may be several minimal flop sequences which obtain a node of the final tree, and we are left with the problem of choosing one of them. At this stage all of them are 'best choices', but at a later stage some of them might turn out to be better choices than others, in the sense that some of them might continue in shorter paths than others. The problem is to decide at this stage which sequences will later on give rise to the shortest paths. A rule guaranteeing success can not be given, but the idea is that those sequences which obtain a node at the highest possible level in the tree are the best choices, for the following reasons. When a node of the final tree is obtained, then 'fixing' the rest of this subtree (i.e. obtaining its nodes in the final tree) involves only the subtree itself, while the rest of the complete tree can be ignored. Thus, in a way, part of the work is split off as an independent subtask of a similar nature. When preferring first to obtain nodes at a high level in the tree, the first subtasks splitting off from the original one, will be the larger ones, while later on smaller subtasks will be split off. This seems to be a more stable approach than splitting off smaller subtasks first.

To illustrate the way in which the above algorithm searches for a transformation sequence, we show in detail some of the algorithm steps in generating a summation formula for the recoupling coefficient (5) shown in figure 1.

Starting from the initial tree $((j_1, j_2)j_6, (j_3, (j_4, j_5)j_7)j_8)j_9$, no single flop sequence obtains a coupled node of the final tree $(((j_1, j_4)j_{10}, (j_2, j_3)j_{11})j_{12}, j_5)j_9$,

however several double flop sequences do, such as :

$$\begin{aligned}
&((j_1,j_2)j_6,(j_3,(j_4,j_5)j_7)j_8)j_9 \\
&\quad\rightarrow\quad (((j_1,j_2)j_6,j_3)k_1,(j_4,j_5)j_7)j_9 \rightarrow ((((j_1,j_2)j_6,j_3)k_1,j_4)j_{12},j_5)j_9, \qquad (10)
\end{aligned}$$

or

$$\begin{aligned}
&\quad\rightarrow\quad (j_1,(j_2,(j_3,(j_4,j_5)j_7)j_8)k_1)j_9 \rightarrow (j_1,((j_2,j_3)j_{11},(j_4,j_5)j_7)k_1)j_9, \qquad (11)
\end{aligned}$$

the latter of which obtains the node j_{11} of the final tree, while the first sequence obtains the node j_{12} (which is the node with $\{j_1, j_2, j_3, j_4\}$ as the set of descendent leaf nodes).

Continuing from the sequence (10), the final tree can be obtained by two more algorithm steps, each consisting of a single flop sequence :

$$\begin{aligned}
((((j_1,j_2)j_6,j_3)k_1,j_4)j_{12},j_5)j_9 &\rightarrow (((j_1,(j_2,j_3)j_{11})k_1,j_4)j_{12},j_5)j_9 \\
&\rightarrow (((j_1,j_4)j_{10},(j_2,j_3)j_{11})j_{12},j_5)j_9.
\end{aligned}$$

The resulting summation formula is a single sum over a product of four 6-j coefficients :

$$\begin{aligned}
&(-1)^{2j_1+j_2+j_5+3j_6+j_7+2j_9-j_{10}} \\
&\times\sqrt{(2j_6+1)(2j_7+1)(2j_8+1)(2j_{10}+1)(2j_{11}+1)(2j_{12}+1)} \qquad (12) \\
&\times\sum_{k_1}(-1)^{k_1+j_1+j_{11}}(2k_1+1)\left\{\begin{array}{ccc} j_7 & j_3 & j_8 \\ j_6 & j_9 & k_1 \end{array}\right\}\left\{\begin{array}{ccc} j_5 & j_4 & j_7 \\ k_1 & j_9 & j_{12} \end{array}\right\} \\
&\qquad\times\left\{\begin{array}{ccc} j_1 & j_2 & j_6 \\ j_3 & k_1 & j_{11} \end{array}\right\}\left\{\begin{array}{ccc} j_{11} & j_1 & k_1 \\ j_4 & j_{12} & j_{10} \end{array}\right\}.
\end{aligned}$$

However, if we continue from the sequence (11), the next algorithm step again requires a double flop sequence to obtain a node of the final tree, e.g. :

$$\begin{aligned}
&(j_1,((j_2,j_3)j_{11},(j_4,j_5)j_7)k_1)j_9 \\
&\quad\rightarrow\quad ((j_1,(j_2,j_3)j_{11})k_2,(j_4,j_5)j_7)j_9 \rightarrow (((j_1,(j_2,j_3)j_{11})k_2,j_4)j_{12},j_5)j_9.
\end{aligned}$$

Continuing from this result, the final tree can be obtained in a single flop sequence. The resulting summation formula is a double sum over a product

of five 6-j coefficients :

$$
\begin{aligned}
&(-1)^{j_2+j_3+j_5+2j_7+3j_9-j_{10}+2j_{11}} \\
&\times\sqrt{2j_6+1)(2j_7+1)(2j_8+1)(2j_{10}+1)(2j_{11}+1)(2j_{12}+1)} \\
&\times\sum_{k_1}\sum_{k_2}(-1)^{k_1+2k_2+j_2+j_8}(2k_1+1)(2k_2+1)\left\{\begin{array}{ccc} j_1 & j_2 & j_6 \\ j_8 & j_9 & k_1 \end{array}\right\}\left\{\begin{array}{ccc} j_7 & j_3 & j_8 \\ j_2 & k_1 & j_{11} \end{array}\right\} \\
&\qquad\times\left\{\begin{array}{ccc} j_7 & j_{11} & k_1 \\ j_1 & j_9 & k_2 \end{array}\right\}\left\{\begin{array}{ccc} j_5 & j_4 & j_7 \\ k_2 & j_9 & j_{12} \end{array}\right\}\left\{\begin{array}{ccc} j_{11} & j_1 & k_2 \\ j_4 & j_{12} & j_{10} \end{array}\right\}.
\end{aligned}
\tag{13}
$$

It is obvious that, in the first step of the algorithm, the sequence (10) obtaining the node j_{12} is a better choice than the sequence (11) obtaining the node j_{11}. Note that the node j_{12} is at a higher level in the final coupling tree than the node j_{11}.

It is also obvious that the single sum formula is optimal, since in the first algorithm step no single flop sequence is successful, which means that at least one summation variable needs to be introduced.

3.2 Traversing a coupling tree

Trying out *all possible single flop sequences* on a coupling tree is done by traversing the tree in a systematic way, starting from the root node, and examining each node when it is visited.

Each coupled node is examined in the following way : On the subtree with this node as the root node, we can consider four possible single flop sequences, i.e. of the following forms :

$$
\begin{array}{ll}
\langle\,((a,b)d,c)f \mid (a,(b,c)e)f\,\rangle, & \langle\,(a,(b,c)d)f \mid ((a,b)e,c)f\,\rangle, \\
\langle\,((a,b)d,c)f \mid (b,(a,c)e)f\,\rangle, & \langle\,(b,(a,c)d)f \mid ((a,b)e,c)f\,\rangle,
\end{array}
$$

where f denotes the current node. For each of these operations we check if it can be performed on the current subtree, i.e. if this subtree 'matches' the appropriate form for the operation. If so, we check if the node e, that is 'created' in the operation, is one of the nodes of the final coupling tree, that has not yet been obtained. If so, a successful single flop sequence is found

and the search process can be stopped, otherwise the search process has to be continued.

This algorithm can easily be extended to *try out all possible N flop sequences*, with a given number N of flops. For that purpose we view an N flop sequence as a single flop sequence followed by an $(N-1)$ flop sequence, a recursive formulation giving rise to the following algorithm. For every coupled node in the tree, we try to find a successful N flop sequence, starting with one of the four possible single flop operations on the current node. This involves checking for each of these four operations if it can be performed on the current subtree, and if so, trying out all $(N-1)$ flop sequences on that subtree (which is done recursively). When $N = 1$, we are at the final stage in the recursive process, where we have to test if the current flop sequence obtains a new node of the final coupling tree. Again, if so, a successful single flop sequence is found and the search process can be stopped, otherwise the search process has to be continued.

A search process traversing a tree, continues until either the examination of the current node is successful or all the nodes have been examined. If all the nodes have been examined without finding a successful flop sequence of given length, then the search is said to be unsuccessful and we have to continue by trying out longer flop sequences.

We are now left with the problem of implementing a search process, which has to examine every node of a tree. Two possible approaches for such a tree traversal are commonly used, i.e. so-called *depth-first* or *breadth-first traversal* [10]. For the problem considered here we will use a breadth-first traversal, which starts from the root node, then considers every coupled node at the first level of the tree, then the nodes at the second level, etc., until every level in the tree is considered. E.g., in the tree $((j_1, j_2)j_6, (j_3, (j_4, j_5)j_7)j_8)j_9$ the coupled nodes at the first level are j_6 and j_8; on the second level the only coupled node is j_7.

When implementing a breadth-first traversal, a problem is caused by the fact that a node in the tree only contains references to its children, while the

information needed is a reference to the next node on the same level in the tree (or to the first node on the next level, in case it is the last node on a level), which is not contained in the node. But this information can be built and stored during the traversal process, in the following way. Assuming that we know which are the nodes on a certain level, we can build a list of nodes which are on the next level, during the process of visiting the nodes on this level; indeed, they are the left and right children of the nodes on the current level. Thus during the traversal process we will keep track of (1) the nodes on the current level which still have to be visited, and (2) the nodes on the next level which are children of visited nodes on the current level and which have to be visited when this level is finished. To store this information we can use the common concept of a *queue*, an abstract datastructure simulating real-life waiting queues, where the first customer in the queue is served first and new customers join at the tail of the queue. At each stage in the traversal the first node is taken from the queue and it is examined. If this is not successful, then the left and right (or right and left) children, if they are coupled nodes, are added at the tail of the queue (to be examined at some later stage). The search continues on the (next) first node in the queue, which will be the next node on the same level in the tree or the first node of the next level in the tree. E.g., a breadth-first traversal, traversing each level from left to right, on the tree $((j_1, j_2)j_6, (j_3, (j_4, j_5)j_7)j_8)j_9$, visiting only the coupled nodes, proceeds as shown in table 2.

step	node	queue
0		(j_9)
1	j_9	(j_6, j_8)
2	j_6	(j_8)
3	j_8	(j_7)
4	j_7	$()$

Table 2: Breadth-first traversal of the tree $((j_1, j_2)j_6, (j_3, (j_4, j_5)j_7)j_8)j_9$, visiting only the coupled nodes.

4 Implementation details

In the program `NJFORMULA` as we present it here, the algorithms for *generating* a summation formula, as described above, are implemented in a set of routines and these are combined in a module, which is used from a main program. The program is designed such that afterwards it can easily be linked with the program `NJSUMMATION` [3], which will supply routines to *evaluate* the formulae generated by the current program.

The module implemented here supplies first of all a function `generate_formula` to generate a summation formula for a given recoupling coefficient. Apart from that it also supplies procedures for the input of a recoupling coefficient in several forms (`read_expr` and `read_njsym`), as well as a procedure `write_formula` for the output of a summation formula.

In this section we describe some implementation details as well as some special features concerning the functionality of these routines. We also discuss in detail a some examples of simple main programs, which illustrate how to use the described routines to generate summation formulae for a number of general recoupling coefficients.

4.1 Datastructures

Throughout the program the index i of a j_i coefficient will be used to identify the j_i coefficient, while a negative index $-i$ will be used to identify a summation variable k_i of the summation formula.

A coupling tree is represented by a datastructure, where each node of the tree is a record structure `NODE`, with a field `index` which contains the index i of the j_i coefficient or $-i$ of the k_i summation variable of the node, fields `left` and `right` containing pointers to the left child and the right child of the node in the tree (leaf nodes, which have no children, contain null pointers), and a field `leafs` containing the set of indices of the j_i coefficients of the leaf nodes which are descendants of this node. E.g., in the coupling tree for the state vector $\mid ((j_1, j_2)j_6, (j_3, (j_4, j_5)j_7)j_8)j_9 \rangle$ the record for the node j_8

contains its index 8, the set $\{3, 4, 5\}$, a left pointer to the node j_3 and a right pointer to the node j_7.

A summation formula is represented by a record structure `FORMULA` containing the following data : 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[i]`, 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; a 6-j coefficient is represented as an array with 6 elements, which are j_i coefficients or k_i summation variables; a j_i coefficient is represented by its index i, while a k_i summation variable is represented by its negative index $-i$. E.g., for the double summation (13), obtained above, the structure contains the following data :

`nrjs`	=	12	`nrks`	=	2	`nrsixjs`	=	5
`jsigns`	=	$(0,2,1,0,1,0,2,1,3,-1,2,0)$				`ksigns`	=	$(1,2)$
`jsqrts`	=	$(0,0,0,0,0,1,1,1,0,1,1,1)$				`ksqrts`	=	$(2,2)$
`sixjs[1]`	=	$(1,2,6,8,9,-1)$				`sixjs[2]`	=	$(7,3,8,2,-1,11)$
`sixjs[3]`	=	$(7,11,-1,1,9,-2)$				`sixjs[4]`	=	$(5,4,7,-2,9,12)$
`sixjs[5]`	=	$(11,1,-2,4,12,10)$						

4.2 Input of recoupling coefficients

Several forms of input for recoupling coefficients are possible. For reasons of compatibility with the programs `NJSYM` [1] and `NJGRAF` [2], we have implemented a procedure `read_njsym`, which reads a recoupling coefficient in the

form used by these programs. It returns the number of j_i coefficients read and the two pointers to the corresponding coupling trees.

However, we also propose another input style for a recoupling coefficient, which is well suited for interactive input. The procedure `read_expr` reads a recoupling coefficient in the form of an expression similar to the mathematical bra-ket notation. It also returns the number of j_i coefficients read and the two pointers to the corresponding coupling trees.

Both input routines read their input from the standard input device `stdin`, which is usually the keyboard. Reading input from a file can be done easily by 'redirecting' the standard input device when executing the program.

4.2.1 Input in the style of `NJSYM`

In the programs `NJSYM` and `NJGRAF` the input of a recoupling coefficient consists of the number of j_i coefficients, the number of couplings, and two lists of triads $(j_{i_1}, j_{i_2}, j_{i_3})$ describing the coupling in the initial and final state vector, where a triad $(j_{i_1}, j_{i_2}, j_{i_3})$ signifies that j_{i_1} and j_{i_2} are coupled to j_{i_3}. To read a recoupling coefficient given in this form, the following actions have to be performed.

In order to build the coupling trees corresponding to this input, the common root node of the trees has to be determined first. We assume that for each state vector the triad describing the root coupling is given as the last triad in the list.

Next a coupling tree has to be built for each of the state vectors. This is implemented in a function `build_tree`, which returns a pointer to the root node of the tree. Building a coupling tree from a list of triads, given the j_i coefficient of the root node, can easily be done by a recursive algorithm, in the following way : When memory space for the node is allocated, the index i of the root node j_i is filled in. Next the list of triads is searched to find the j_{i_1} and j_{i_2} coefficients that are coupled to j_i. If no such coefficients are found, then j_i is a leaf node, pointing nowhere left and right and the set of

leaf nodes is simply the singleton $\{j_i\}$. Else j_i is a coupled node, pointing left to a subtree with j_{i_1} as the root node and pointing right to a subtree with j_{i_2} as the root node. The building of these subtrees is done in the same way, i.e. by two recursive calls to the function `build_tree`, returning the pointers to be assigned to the fields `left` and `right` of the node j_i. The set of leaf nodes for j_i is obtained as the union of the set of leaf nodes of its left and right subtrees j_{i_1} and j_{i_2}.

4.2.2 Input in the form of expressions

Another representation of recoupling coefficients, well suited for input and output, is an expression, which closely resembles the bra-ket notation of a recoupling coefficient. The coupling of j_{i_1} and j_{i_2} to j_{i_3} is expressed as $(i_1, i_2)i_3$. In their turn j_{i_1} and j_{i_2} can be the couplings of other j_i coefficients, which is expressed in the same way. This gives rise to a string consisting of nested couplings, describing the coupling tree for a state vector. Two such state vector expressions, enclosed in brackets (`<`) and (`>`) and separated by a vertical bar (`|`), then form the input expression for a general recoupling coefficient. E.g., the input expression for the recoupling coefficient (5) is :

```
< ((1,2)6,(3,(4,5)7)8)9 | (((1,4)10,(2,3)11)12,5)9 > .
```

The handling of such an input expression is implemented in the procedure `read_expr`.

A state vector expression can be read and transformed into a coupling tree by means of a recursive algorithm, which we have implemented in a function `read_tree`. To that purpose, we formally define a state vector as either a positive number, or an expression of the form (*state vector* , *state vector*) *number*. The latter is a recursive definition, saying that a state vector can consist of two state vectors, separated by a ',' and enclosed in brackets, followed by a number. The number is the index of the root node in the coupling tree of the state vector, while the two state vectors are its left and right children. From this definition the algorithm to build a coupling

tree corresponding to a given state vector expression, which can be viewed as a string of tokens, follows easily. A general step in the algorithm works on an expression corresponding to a subtree, starting from a given node, and builds this subtree in the following way : Memory space is allocated for the root node of the subtree, a pointer to which will be returned by the algorithm when it finishes. If the next token in the input string is a number, then this subroot node is a leaf node, for which the index is this number, the set of leaf nodes is the singleton containing this number and the left and right pointers are set to null. Else, if the next token in the input string is an opening bracket '(', then this subroot node is a coupled node. This means that the next part of the input string is an expression corresponding to its left child subtree, which can be built by a recursive call to the same algorithm. After that the next token in the input string must be a comma and the next part will be an expression corresponding to the right child subtree of the subroot node, again to be built recursively. Next in the input string will be a closing bracket ')', followed by a number, which is the index of the subroot node. Finally the set of leaf nodes of the subroot node is obtained as the union of the set of leaf nodes of the left and right subtrees.

During the recursive process the input is checked for errors, such as forgotten commas or non-matching brackets, in which case we finish the recursive process and report it to be unsuccessful. In this implementation the recursive function `read_tree` is called twice from the procedure `read_expr`, to read the bra-vector and the ket-vector of the recoupling coefficient. Also here the input is checked for errors; if errors occur, the procedure allows to retry the input of an expression, until a correct input expression is given.

Another feature we have implemented, to allow simplification of the input expression, is automatic numbering of the coupled nodes. In that case only the indices of the leaf nodes have to be given explicitly, while for the coupled nodes an index is generated automatically during the reading process by means of consecutive integers. E.g., for the recoupling coefficient (5) the simplified input expression is

```
< ((1,2),(3,(4,5))) | (((1,4),(2,3)),5) >
```

and the indices for the coupled nodes generated by the program will be the same as in the input expression given above. This automatic numbering is only possible if the number of leaf nodes is given first. The option to use automatic numbering or not is left to the user and is controlled by a global variable `AUTONUM`, which can be set to `ON` or `OFF` (by default it is set to `ON`).

For a better readability of an input expression, we also allow separators (such as spaces, tabulators and newline characters) to be put arbitrarily in between the tokens.

4.2.3 Remark

When a recoupling coefficient is given, it can happen that some of the coupled nodes in both trees are the same, in the sense that they have the same set of leaf nodes, while the index i of the j_i coefficient for the nodes is different in the two trees. In such cases the value of the recoupling coefficient will be trivially zero, if the corresponding j_i coefficients are not the same. To avoid unnecessary computations (in evaluating the summation formula, or sometimes even in generating a summation formula), we want to report on these cases when reading a recoupling coefficient, during which process they can easily be detected. For that purpose we build a list of delta functions $\delta(j_{i_1}, j_{i_2})$, that is returned as an extra parameter by the input routines. In the program presented here we will not yet use this information, as here we are mainly dealing with the algorithm for the generation of a summation formula, independent of the actual j_i values. But the program `NJSUMMATION`, which will deal with the actual j_i values, will indeed need this information to check if any computations are actually needed for a given recoupling coefficient. For that reason we have already implemented this feature in the input routines.

4.3 Generation of a summation formula

The function `generate_formula` searches a transformation sequence for a recoupling coefficient, given by the corresponding coupling trees and the

number of j_i coefficients, and returns a pointer to the generated summation formula. If for some reason the search was not successful, then a null pointer is returned. It is a rather straightforward implementation of the algorithms described in section 3.

The main body is a loop, which searches in each step for a minimal flop sequence obtaining one more coupled node of the final coupling tree, until all the nodes of the final coupling tree are obtained. Each step in the main loop is in its turn a loop trying out all N flop sequences, for $N = 1, 2, 3, \ldots$, until a successful flop sequence has been found. Trying out all N flop sequences, for given N, is implemented in a recursive algorithm as described earlier, examining every node of the coupling tree in a breadth-first traversal. For practical purposes we have set an upper limit for the values of N to be tried out, which is kept in a constant `MAXFLEN` and is here set to 5 (in the examples described below no higher value was needed).

To keep track of which nodes of the final coupling tree are already obtained, we make use of the field `leafs` in each node, defined to contain the set of descendent leaf nodes of the node. When a node of the final coupling tree is obtained, we set its field `leafs` to be the empty set, thus making clear that in a next step one should not try to obtain this node anymore. When all the nodes of the final coupling tree have their leafset set to empty, then the transformation process is finished. Before starting the transformation process, a sweep of the final coupling tree is performed, checking for each node if it already occurs in the initial coupling tree (by checking for equal leafsets). If so, for such a node in the final tree, the field `leafs` is set to empty. The root node will be such a node.

In each step the contribution of the successful flop sequence is recorded in the datastructure that will finally contain the complete summation formula. For every flop operation in the sequence, the appropriate entries in the `jsigns`, `ksigns`, `jsqrts` and `ksqrts` arrays are updated accordingly, and a new 6-j coefficient is added. Every exchange operation in the sequence will influence some entries in the `jsigns`, `ksigns`, `jsqrts` and `ksqrts` arrays.

Apart from updating the summation formula in generation, the basic operations in the successful sequence also involve updating of the current coupling tree in each step. In an exchange operation the left and right children of a node are interchanged, which can be done easily by interchanging the `left` and `right` pointer in the node. A flop operation of the form $\langle ((a,b)d,c)f \,|\, (a,(b,c)e)f \rangle$, working on the current node f, changes the structure of the subtree more drastically : the left child d is replaced by its own left child a, while the right child c is replaced by a new node e, which has the node c as its right child and the node b as its left child. This transformation involves only updating of the appropriate pointers, no copying of subtrees is necessary. The other flop operations involve similar actions.

Finally we would like to make the following remark. Trying out several examples, we noticed that in some cases a different result was obtained when traversing the tree levels from left to right or from right to left. Also trying to transform the final tree into the initial tree instead of vice versa, sometimes produced a better result. For that reason, the function `generate_formula` tries out several of these possible search processes, and keeps the best result that is obtained in this way. The number of possibilities tried out is controlled by a global variable `SEARCH`, which can be set by the user to either `SHALLOW`, `DEEP` or `VERYDEEP`; a typical choice is `DEEP`, which is also the default value. No matter what the value of `SEARCH` is, the search always stops if a formula with less than two summation variables is obtained, since such a formula is guaranteed to be optimal.

For debugging purposes we have also implemented an option to let the program output some intermediate results, which can be used to follow the generation process. This debug information is written to a file, the name of which can be chosen. An overview of each search process is given as a list of coupling trees (in the form of an expression) obtained in each step of a search process, as well as the number of flops in the sequence obtaining the current coupling tree in a step. For each search process the obtained formula is also given. Debugging can be initialised by calling the procedure

`init_debug`, giving the filename as a parameter, and must be closed by calling a procedure `close_debug`.

4.4 Main program

The following simple main program can be used to generate summation formulae for an arbitrary number of recoupling coefficients, which are read interactively in the form of expressions. The headerfile `njformul.h` contains a set of declarations, which are needed by the program. The global variable `AUTONUM` is initialised to `ON`, meaning that only leaf nodes of the recoupling coefficients will be given explicitly. The global variable `SEARCH` is initialised to `DEEP`, signifying that both traversing left-to-right and right-to-left will be tried out in a transformation of bra to ket. The `do...while` loop reads in each step a new recoupling coefficient, generates the corresponding summation formula and writes this formula. This loop continues until an answer `n` or `N` is given to the question `Continue(y/n)?`.

```
#include "njformul.h"

main ()
{
        int nrjs;
        NODE *bra, *ket;
        DELTAS delta;
        FORMULA *form;
        char ch;

        AUTONUM = ON; SEARCH = DEEP;
        do {
          read_expr (&nrjs, &bra, &ket, &delta);
          form = generate_formula (nrjs, bra, ket);
          write_formula (form);
          printf ("Continue (y/n) ? "); scanf ("%c", &ch);
        } while (ch != 'n' && ch != 'N');
}
```

The following main program reads one recoupling recoupling coefficient in

the style of NJSYM and writes the generated formula. Debugging is initialised and the debug information will be written to a file formula.deb. The variable SEARCH is set to VERYDEEP, meaning that both left-to-right and right-to-left traversal will be tried out, for transforming bra to ket as well as ket to bra. The variable AUTONUM is not significant in this case.

```
#include "njformul.h"

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

        init_debug ("formula.deb"); SEARCH = VERYDEEP;
        read_njsym (&nrjs, &bra, &ket, &delta);
        form = generate_formula (nrjs, bra, ket);
        write_formula (form);
        close_debug ();
}
```

5 Results and discussion

The program NJFORMULA 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 [11], and a Sun Sparc running Unix) using several compilers (such as Turbo C++ [12], Gnu CC [13] and SPARCompiler C). Since this program will be joined later with the program NJSUMMATION, which will include a parallel implementation on a T800 transputer system (hosted by PCs under MS-DOS), we have also tested the program NJFORMULA on this transputer system, using the 3L Parallel C compiler [14]. Of course the (sequential) program uses none of the parallel features of this compiler, and it is executed on a single transputer.

To compare the results obtained by our program NJFORMULA with the

results obtained by the program NJGRAF, we have compiled the program NJGRAF (omitting the calls to the subroutine GENSUM) on the Sun Sparc using the Sparc Fortran-77 compiler and on the transputer system using the 3L Parallel Fortran compiler [15].

First we want to compare our results for the test cases 1, 2 and 4 discussed by Bar-Shalom and Klapisch [2], with the results of their program NJGRAF. These test cases correspond to the following recoupling coefficients (note that (G_1) corresponds to the recoupling coefficient (2)) :

$$(G_1)\ \langle\, ((j_1,j_2)j_5,(j_3,j_4)j_6)j_7 \mid (j_1,((j_2,j_3)j_8,j_4)j_9)j_7 \,\rangle$$

$$(G_2)\ \langle\, (((j_1,j_2)j_8,((j_3,j_5)j_9,(j_6,j_7)j_{10})j_{11})j_{12},j_4)j_{13} \mid (((j_1,j_2)j_{14},((j_3,(j_7,j_5)j_{15})j_{16},j_6)j_{17})j_{18},j_4)j_{13} \,\rangle$$

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

Table 3 shows for each test case the number of angular momenta involved, the number of summation variables and the number of 6-j coefficients in the generated summation formula, for both programs NJGRAF and NJFORMULA. To give an idea of the complexity of the search process in the program NJFORMULA, we also give the lengths of the minimal flop sequences obtained in each step of the generation algorithm. For each case NJFORMULA obtains a summation formula which is as good as the one obtained by NJGRAF, i.e. a formula with the same number of summation variables and 6-j coefficients. Running times of both programs are comparable and are of the order of 0.01 s on the Sun Sparc and on the 486 PC (under Linux).

Apart from these test cases described in [2], we also discuss the results for some other recoupling coefficients, some of which give rise to quite a complicated search process :

$$(F_0)\ \langle\, (j_1,j_2)j_5,(j_3,j_4)j_6)j_7 \mid ((j_1,j_3)j_8,(j_2,j_4)j_9)j_7 \,\rangle$$

$$(F_1)\ \langle\, ((j_1,j_2)j_6,(j_3,(j_4,j_5)j_7)j_8)j_9 \mid (((j_1,j_4)j_{10},(j_2,j_3)j_{11})j_{12},j_5)j_9 \,\rangle$$

$$(F_2)\ \langle((j_1,(j_2,j_3)j_7)j_8,(j_4,(j_5,j_6)j_9)j_{10})j_{11} |(((j_1,(j_4,j_5)j_{12})j_{13},j_2)j_{14},(j_3,j_6)j_{15})j_{11}\rangle$$

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

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

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

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

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

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

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

Table 3 also shows the results obtained for these recoupling coefficients, for both programs `NJGRAF` and `NJFORMULA`. For the simplest cases, i.e. (F_0)–(F_6), the running time of both `NJFORMULA` and `NJGRAF` is again of the order of 0.01 s. For some of these cases, i.e. (F_0), (F_2), (F_3) and (F_6), both programs obtain a similar formula, with the same number of summation variables. However, for several cases, i.e. (F_1), (F_4) and (F_5), `NJFORMULA` obtains a better formula than `NJGRAF`, with less summation variables. Note that (F_1) corresponds to the recoupling coefficient (5) and that `NJFORMULA` obtains the single sum (12), while `NJGRAF` obtains a double summation similar to formula (13). For (F_5) we even get a double summation formula, where `NJGRAF` obtains a 4-fold summation.

For the most complicated cases, i.e. (F_7)–(F_9), `NJFORMULA` still generates a summation formula in very little time: (F_7) takes about 0.02 s, while (F_8) takes 0.15 s and (F_9) takes 0.50 s. However for these cases `NJGRAF` gives no result at all : the program crashes during the search process.

From these examples it is clear that our program `NJFORMULA` performs very well. The results are at least as good as the results of the program `NJGRAF` and are obtained in a comparable running time. In many cases

		NJGRAF		NJFORMULA		
Test case	#j's	#k's	#6-j's	#k's	#6-j's	length of sequences
(G_1)	9	0	2	0	2	1-1
(G_2)	18	0	2	0	2	1-1
(G_4)	30	3	11	3	11	1-2-1-1-2-1-2-1
(F_0)	9	1	3	1	3	2-1
(F_1)	12	2	5	1	4	2-1-1
(F_2)	15	1	5	1	5	1-2-1-1
(F_3)	15	2	6	2	6	2-2-1-1
(F_4)	15	3	7	2	6	2-2-1-1
(F_5)	18	4	9	2	7	2-2-1-1-1
(F_6)	18	2	7	2	7	1-2-2-1-1
(F_7)	18	–	–	4	9	3-2-2-1-1
(F_8)	21	–	–	7	13	3-4-1-3-1-1
(F_9)	27	–	–	9	17	3-2-4-4-1-1-1-1

Table 3: Results obtained by `NJGRAF` and `NJFORMULA` for a range of recoupling coefficients.

we obtain a better formula than `NJGRAF`, and `NJFORMULA` can also handle very complicated cases where `NJGRAF` crashes. It is a small program (the executable code is about 20KB, while the compiled code of `NJGRAF` is 260KB), which runs very well on various systems, also on small systems such as PCs.

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

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

[3] V. Fack, S.N. Pitre and J. Van der Jeugt, *New efficient programs to calculate general recoupling coefficients, part II (in preparation)*

[4] 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).

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

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

[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] E. Horowitz and S. Sahni, Fundamentals of Data Structures in Pascal (Pitman, London, 1984).

[11] 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`.

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

[13] 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`.

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

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