DFPD/93/TH/76

hep-th/9401082

`WBase`: a C package to reduce tensor products of Lie algebra representations

Antonio Candiello*
Dipartimento di Fisica, Università di Padova
Istituto Nazionale di Fisica Nucleare, Sezione di Padova
Italy

Abstract

It is nearly twenty years that there exist computer programs to reduce products of Lie algebra irreps. This is a contribution in the field that uses a modern computer language ("C") in a highly structured and object-oriented way. This gives the benefits of high portability, efficiency, and makes it easier to include the functions in user programs. Characteristic of this set of routines is the *all-dynamic* approach for the use of memory, so that the package only uses the memory resources as needed.

*Supported in part by M.P.I. This work is carried out in the framework of the European Community Programme "Gauge Theories, Applied Supersymmetry and Quantum Gravity" with a financial contribution under contract SC1-CT92 -D789.

PROGRAM SUMMARY

Title of program: `WBase`

Catalogue number:

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

Licensing provisions: none

Computer: Vax 4000/90, DecSystem 5500; *Installation:* Department of Physics, Padua, Italy

Operating system: Vms 5.5-2, Ultrix V4.3A

Programming language used: Vax C without any extension

Memory required to execute with typical data: the program uses dynamic allocation, the memory used can be just a few words up to the available memory on the hardware

No. of bits in a word: 32

No. of lines in distributed program, including test data, etc: 875 lines

Keywords: Lie algebras, tensor products, irreducible representations

Nature of physical problem
Irreducible representation manipulations are needed in a wide number of physical arguments, such as high energy particle physics, supergravity theories, grand unification models

Method of solution
The Dynkin technique for manipulating Lie algebra irreps is used. A sophisticated dynamic allocation scheme let our program to use memory only when really needed

Restrictions of the complexity of the problem
The limit on the application of our program is related to the dimension of the irreducible representations needed when computing tensor products; this limit is dependent on the hardware, as dynamic allocation does not set software bounds to the size of data

Typical running time: 2.41 s for the test run

References

[1] B. W. Kernighan and D. M. Ritchie, "The C Language" (Prentice-Hall, 1978.

1 Introduction

The necessity of group representation theory in several sectors of physics is well known in the community. In particular, in particle physics, thanks to the grand unification models, it is now nearly a necessity to work out some group representation manipulations. Another sector where representation theory is heavily used is the construction of supergravity theories; this is in particular a sector in which we are involved and the main reason for us for constructing such a package.

Consulting standard references on the subject [7, 3, 2], in particular [7], one finds tables of dimensions of irreducible representations (irreps) and tables with the reduction of tensor products of irreps. But in certain cases cases the tables are not large enough, and then we have two possible choices: a) do the calculations needed with tools like Young tableaux; b) search for a computer program that does the job. Opting for the second choice (a necessity for some Lie groups and for irreps above certain dimensions), one can refer to the works of Patera et al [1]. Our philosophy has been to restart with the C language [6] and, thanks to the C powerful dynamic data allocation facilities, to keep all needed data structures run-time allocated; we also decided to make use of the most modern object-oriented techniques in order to free the user of our package from the complexities of the dynamic allocation scheme, and to keep the source code clear and easily modifyable (see, for example, [4, 5]). The advantages are:

- dynamic allocation: the routines work on all machines, and memory is used only when really needed; this is a necessity, as memory requirements can vary from a few bytes to several megabytes according to the tensor products needed. Dynamic allocation gives us also the possibility to restart the routines changing the algebra without exiting the program, permitting the use of this package as the engine in an irrep products table generator;

- object-oriented technology: it is thanks to this approach that we can

keep tractable such a system with continuous allocations and deallocations of memory, and as an added advantage we can give the user a consistent and easy interface to our routines.

As an example we cite the clarity of this fragment of C source code:

```
pcurr pc; wplace *p;
wblock *base;
for(p=pstart(base,&pc);p!=NULL;p=pnext(&pc))
  wdisp(p->vect);
```

where the entire irrep weight system, pointed by `base`, is displayed to the standard output. Note that there is nothing here to know about the internal data structure of `base`.

The work is organized as follows: in section 2 we describe some of the intricacies of the Dynkin view of representation theory, where the fundamental object is the "weight vector"; section 3 is devoted to the data structures, their links and allocation/deallocation schemes. In section 4 we enter in the core of the package, describing the higher level routines which

- compute the dimension of an irrep
- compute the casimir index of an irrep
- generate the weight system of an irrep
- compute the degeneration of each weight in the weight system
- gives the reduction of tensor products of two given irreps.

Section 5 describes the user-interaction routines and a simple session at the terminal; we stress that these interaction routines can be easily substituted by more sophisticated ones, for example batch-oriented routines to generate full tables, or GUI- (Graphical User Interface) oriented routines to simplify the task of using the routines to the casual user. In section 6 we discuss the performance of the package for both the points of view of 1) speed and 2) memory; they are somewhat connected in that, due to our linked list scheme,

more used memory means more time to process the data. In this last section we also discuss the direction of future improvements of the package for both problems of memory and speed.

2 Dynkin's approach to representation theory

We will try to keep this section as short as possible, focusing mainly on the algorithms needed for our purposes; all material is derived from [7], to which we refer also for a comprehensive introduction to the subject.

As is well known, a simple Lie algebra is described by his set of generators which have the commutation rules

$$[T_i, T_j] = f_{ijk} T_k, \qquad i, j, k = 1, \ldots, d \tag{1}$$

where the f_{ijk} are the algebra structure constants; a convenient basis for the generators is the Cartan-Weyl basis, where the T_i are subdivided in the l simultaneously diagonalizable generators H_i,

$$[H_i, H_j] = 0 \qquad i = 1, \ldots, l \tag{2}$$

and the remaining generators E_α,

$$[H_i, E_\alpha] = \alpha_i E_\alpha, \qquad i = 1, \ldots, l;\ \alpha = -\frac{d-l}{2}, \ldots, \frac{d-l}{2} \tag{3}$$

so that the structure constants are now organized as a set of l-vectors α: l is the *rank* of the algebra, and the set of l-vectors α are the *roots*. It results that all roots can be constructed via linear combinations by a set of l roots, called *simple roots*.

With the simple roots one then constructs the $l \times l$ *Cartan matrix* which is needed in our package to construct the weight system of a given irrep, as we will see; the Cartan matrix is the key to the classification of the Lie algebras, and it is known for all of them: the A_n, B_n, C_n, D_n series and the exceptional algebras G_2, F_4, E_6, E_7, E_8. The other matrix needed is the *metric* G_{ij}, which is related to the inverse of the Cartan matrix.

The Dynkin approach is so powerful because the Cartan matrix is all we need to completely describe the algebra. This is at the basis of our

package: the routine `WStartup`, given the name of the algebra[1], takes care of generating algorithmically the related Cartan matrix, along with the metric and two weight vectors we need also.

The metric matrix (which we call `wmetr`) introduces a scalar product in the space of *weight vectors*, which are l-uples of integer numbers describing either irreps or the different states in a given irrep. Each irrep in an algebra is uniquely classified by such an l-uple, the *highest weight* whose components are all positive integers (the Dynkin labels); both the weight vectors and the highest weight of an irrep are associated to the typedef `wvect`.

The dimension of an irrep Λ can be calculated with the help of the *Weyl formula*, which is

$$\dim(\Lambda) = \prod_{pos.roots\alpha} \frac{(\Lambda + \delta, \alpha)}{(\delta, \alpha)} \tag{4}$$

where Λ is the highest weight determining the irrep, δ is a special vector which has Dynkin components which are all equal to unity, and $(\ ,\ )$ is the scalar product constructed with the metric G_{ij}. The *positive roots* can be obtained with a specific algorithm which will be explained below in the context of weight systems: specifically, we derive the weight system of the adjoint irrep keeping only the positive weight vectors (the upper half of the spindle). The weyl formula is implemented by the `weyl` function.

Sometimes it is necessary to know other invariants of the irreps, apart from the dimension; we have implemented the second order casimir invariant

$$\mathrm{cas}(\Lambda) = \frac{\dim(\Lambda)}{\dim(\mathrm{adj})} \tag{5}$$

which is calculated by the function `casimir`.

The *weight system* of an irrep is made up by a sequence of weight vectors, which describe all the states, degenerated or not, in an irrep; the weight system is also needed to work out products of irreps, which is the main task of our package. The weight system is obtained by subtracting rows of the Cartan matrix from the highest weight as described by the following procedure:

start with the highest weight: $u := \Lambda$*;*

[1] actually only the A_n, B_n and D_n series, through `Afill`, `Bfill` and `Dfill` are implemented, but it is really a simple task to add support for the other algebras.

for all i in $1, \ldots, l$
subtract from u the i-th row of the Cartan matrix once, twice, ..., u[i] times to give the vectors $u_1, \ldots, u_{u[i]}$;
add all u_k *to the weight system*
restart the procedure with $u := u_k$

For example, let us see what happens for the **3** of $SU(3)$: given the Cartan matrix $\begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix}$, and the highest weight (0 1) of the **3** irrep, we have:

(0 1) $\rightarrow \boxed{(0\ 1)}$

- first component $= 0 \Rightarrow$ no subtractions of (2 -1)
- second component $= 1 \Rightarrow$ one subtraction of (-1 2): it gives (0 1)−(-1 2)=$\boxed{(1\ -1)}$;

  (1 -1) $\rightarrow$
  - first component $= 1 \Rightarrow$ one subtraction of (2 -1): it gives (1 -1)-(2 -1)=$\boxed{(-1\ 0)}$

    (-1 0) $\rightarrow$ *no further subtractions*
  - second component $= -1 < 0 \Rightarrow$ no subtractions of (-1 2).

So we have the weight system

$$\begin{bmatrix} 0 & 1 \\ 1 & -1 \\ -1 & 0 \end{bmatrix}. \tag{6}$$

For higher ranks and/or weights the weight system grows rapidly, but the steps are essentially the same. The routine that takes care of constructing the weight system is `wtree`; it is a recursive routine that uses the procedure described.

The trickier part of the construction of the w.s. results from the fact that each weight vector in the w.s. can be degenerated; the degeneration is calculated using the *Freudenthal recursion formula*, level by level, knowing the degeneration of previous levels: it reads

$$\deg(w) = \frac{2 \sum_{pos.roots \alpha} \sum_{k>0} \deg(w + k\alpha)\,(w + k\alpha, \alpha)}{\| \Lambda + \delta \|^2 - \| w + \delta \|^2} \tag{7}$$

with the initial condition $\deg(\Lambda) = 1$. This calculation is performed by the routine `freud` for each weight vector.

Once developed the machinery to generate full weight systems along with their degenerations, we can reduce the products of irreps, i.e. solve for the R_i in the equation

$$R_1 \otimes R_2 = \bigoplus_i R_i; \tag{8}$$

clearly, as each irrep is individuated by a specific highest weight, this statement can be re-expressed by saying that, given two highest weight vectors, we have to find the list of h.w. vectors, each with its degeneration. The algorithm is simple, but requres many computations; here we present the main steps along with the related routines called:

- (`wsyst`): generate the w.s. of R_1;
- (`wsyst`): generate the w.s. of R_2;
- (`bprod`): construct the set $\{w_1 + w_2\}$, $w_1 \in$ w.s.(w_1), $w_2 \in$ w.s.(w_2); repeat until the set is empty:
  - (`whighest`): find the h.w. w of the set (it is the weight vector w which gives the maximum $\overline{R} \cdot w$, where $\overline{R}$ is a known vector specific to each algebra which we call `whigh`).
  - (`wsyst`): generate the w.s. related to w;
  - (`bremove`): subtract from the set each vector in this w.s.

This is precisely the sequence followed by the routine `wpdisp`.

3 Dynamic data structures

The fundamental data structure underlying our routines is the weight vector, which consists of an l-uple of small (small, as the weight vector components grow slowly with respect to the dimension of the irrep) integer numbers. In a conservative approach one would use the following definition,

```
#define MAX 10
typedef int wvect[MAX];
```

to introduce the related weight vector data type. However, this definition is redundant: firstly we can note that an `int` is really more than needed, it suffices to use the `char` type for the components of the weight vector. Secondly, the fixed dimension of `wvect` really waste memory when the rank l is less than `MAX` and, on the other hand, it limits the maximum rank to `MAX`, so we will need to recompile to use larger rank algebras.

To overcome these drawbacks, we use the dynamic allocation features of the C language, defining

```
typedef char *wvect;
```

and introducing the related allocation/deallocation routines

```
wvect walloc();
void wfree(wvect w);
```

which alloc and free a vector of length `wsize`$\equiv l$ which can be changed at run-time.

For weight vectors embedded in the weight system of an irrep, however, we would require also some other information, such as the degeneration and the level of the weight vector, so we are led to an extended weight structure which we call `wplace`:

```
typedef struct tplace {
 tdeg deg;
 tlevel level;
 char vect[1];} wplace;
typedef short tlevel;
typedef unsigned short tdeg;
```

where we keep separated the `typedef`s for `deg` and `level` in order to have easy expandibility (the `short` for `tlevel` works well for irreps with height less than nearly 32000). The `wplace` data type is used essentially to give a structure to the raw data kept in the blocks on which we base our granularity-flexible allocation scheme; also, `wplace` is the standard object managed by the routines `pstart`/`pnext`/`plast` which hide the dynamic block allocation scheme.

3.1 Mid-level routines

In order to manipulate the lists of blocks that contain the weight systems all that the user needs is a pointer, say `base`, of type `wblock`, that contains the weight system of the given irrep. Then it suffices to:

1. declare an object of type `pcurr` (say, `pcurr pc;`)
2. use the iteration functions `pstart`/`pnext` as in

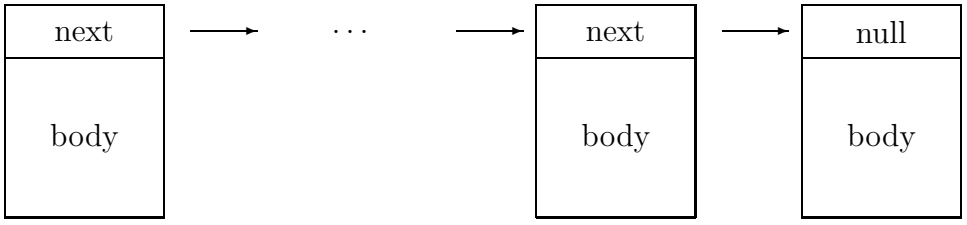
   ```
   for(p=pstart(base,&pc);p!=NULL;p=pnext(&pc))
     < do something with p >
   ```

   remembering that: `p->vect` gives the weight vector, `p->deg` its degeneration, and `p->level` the level of the vector within the weight system;
3. to remove the last entry from the list one uses `base=plast(p,base)`, with the just removed vector returned in the area pointed by `p`;
4. the construction of the w.s. list will be normally handled by the higher level routines to be described in section 4.

3.2 Inner data structure

The difficulty in constructing weight systems of arbitrary irreps is due to the fact that the dimension of the table needed to store the weight system is not known in advance. The only solution that does not waste large amounts of memory and does not limit our routines more than the hardware does is a multiple block allocation scheme.

The data structure is as follows:

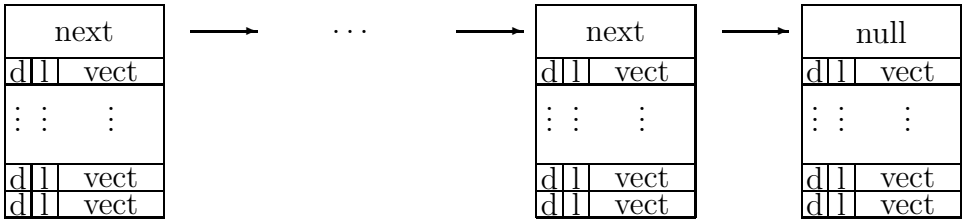


a singly linked list of blocks of identical size fixed at run-time (according to the fragmentation required) by the number of `wplace` vector entries as

given in `bsize`; the typedef that defines a single block leaves therefore its main structure undefined, as it is different for different ranks `wsize` and block dimension `bsize`:

```
typedef struct tblock {
 struct tblock *next;
 char body[1];} wblock;
```

also this definition, as the `wplace` one, is used through casts that give form to unstructured raw data as returned by the allocator `balloc`. Single blocks are freed with a call to `bfree`, linked blocks are freed with a call to `bsfree` acted on the first of them. The structured form of the blocks, when casting with `wplace` and defining `bsize` and `wsize`, is as follows:



4 High level routines

Given the underlying data structure, the task of handling the lists of vectors is taken by a few routines:

- `wsave(w,base,level)`:

  insert in the list pointed by `base` the weight vector `w` before any weight vector at the same or higher level, if `w` is not already present;

- `wremove(w,base)`:

  remove a weight vector `w` from the list `base`, counting the degeneration, i.e. by decrementing the degeneration associated, if greater than 1, and removing the vector only if the degeneration is just 1;

- `wtree(w,base,level)`:

construct, repeatedly calling `wsave` and recursively itself, the weight system vectors of highest weight `w` with the algorithm described in section 2, without computing the degeneration;

- `bfreud(base)`:

  calculate the degeneration for each element in the list through the Freudenthal formula coded in the function `freud` called for each element of the weight system in `base`.

These last two functions are called in turn by `wsyst(hw)`, which return the full linked list of blocks of the weight system of highest weight `hw` along with the degenerations.

Beyond `wsyst`, there is a set of routines to manipulate irreps:

- `weyl(hw)`, `wdim(base)`

  both return the dimension of a given irrep, but `weyl` uses the Weyl formula, while `wdim` requires the construction of the w.s. via `wsyst` (`wdim` is really a test routine);

- `casimir(hw)`, `wind(base)`

  both return the index of a given irrep, but `casimir` uses a formula which involves only the highest weight `hw` and the positive roots, like the Weyl formula, while `wind` requires the construction of the w.s. via `wsyst` (also `wind` is really a test routine);

- `wheight(hw)`, `whigher(base)`

  `wheight` returns the height of the irrep of highest weight `hw`, while `whigher` returns the higher weight vector in `base`.

The functions `weyl`, `casimir`, `wheight` are called by `wfdisp(hw)` to give the information about the irrep with h.w. `hw`.

- `wpdisp(hw1,hw2,mod)`

  this function hides all the complexities of reducing products of irreps and of the underlying data structure, by giving to the standard output all the irreps in the product, from the highest to the lowest, according to the modality choosen by `mod`[2]. The product routines are also

[2]See the header file `wbase.h`.

available as iteration functions `wpstart(b)`/`wpnext(b,base)`, similar to `pstart`/`pnext`, giving the irreps contained in the product of each iteration as `wblock` data types.

5 User interaction

The convenience of the object-oriented approach is visible in the simplicity of the setting up of a one-page file which contains all that is necessary for connecting our routines with the user.

This file is `wmain.c` and can be separately compiled; a more sophisticated user interaction interface may be constructed taking this file as an example. Let us describe the fundamental routines that need to be called to build up such a system:

- `wstartup(type,rank)`, `wcleanup(string)`

  these are the open/close routines for setting up the algebra; given e.g. the algebra B5 (SO(11)) one would call `wstartup('B',5)` to allocate the related vectors and matrices, and then `wcleanup()` to deallocate them. Each invocation of `wstartup()` must be followed by `wcleanup()`. Naturally, the construction of a table of several algebras would call for multiple invocations of this pair of routines.

- `wread(hw)`, `wfdisp(hw)`:

  these are the input/output routines, which read a highest weight vector from the standard input and display the related irrep information on the standard output. One has to remember that these routines need a valid `wvect` which is handled by `walloc()`/`wfree()`.

- `wsyst(hw)`, `bdisp(base)`, `bsfree(base)`:

  these routines manage the weight systems:

  `base=wsyst(hw)` generates the w.s. of `hw` and returns the related `wblock` pointer; `bdisp(base)` displays to the standard output all entries of the w.s., and `bsfree(base)` deallocates the complete linked list of pointers;

- `wpdisp(hw1,hw2,mod)`:

this single routine manages all the intricacies connected to the reduction of products of irreps, displaying to the standard output all the representations contained in "`hw1` $\otimes$ `hw2`", according to the `mode` switch: if the `WTREE` flag is switched on, the full tree of the found irreps it si showed to the standard output, if the `WFAST` flag is switched on a faster and imprecise method of generating products of irreps is used which does not give correctly the degenerations.

The more sophisticated user of our package would need the access to the iteration functions `wpstart`/`wpnext` which make the skeleton of `wpdisp`; essentially, `wpdisp` is structured as follows:

```
buf=bprod(wsyst(hw1),wsyst(hw2));
for(b=wpstart(buf);b!=NULL;b=wpnext(b,buf))
  ;
```

where `bprod` takes care of constructing a combined w.s. made from sums of those of its arguments. The reduction process is hidden by the iteration routines `wpstart`/`wpnext`, which essentially at each step find the higher weight in `buf`, generate his w.s., display some information on it, and remove it from `buf`.

Let us now turn on the use of our package as it is; it is very simple: after running the program (wbase) you will be requested for the algebra. After entering, for example, D5≡SO(10), a little menu with the possible operations will be presented ('P' for product, 'R' for informations on irreps,...). Selecting 'P', the user will be prompted for two couples of 5 integer numbers each (the h.w. of the irreps to be multiplied). With the following input (corresponding to the **10** and **16** irreps of D5, respectively),

```
1 0 0 0 0
0 0 0 0 1
```

we would have the output

```
(D=144,C=68,H=18)  HW=(1 0 0 0 0)
(D=16,C=4,H=10)    HW=(0 0 0 0 1)
```

which means $10 \otimes 16 = 144 \oplus \overline{16}$. To associate h.w. to dimensions, casimirs and levels of the irreps it is useful to choose 'R' as option. On VMS systems ctrl-z returns the program to the main menu; on the others, EOF will force the exit.

6 Performance, memory requirements and future developments

The package is now, we think, useful for the standard irrep-related necessities of physicists, as the routines are reasonably fast and the memory requirements kept to a minimum such as to permit working also with small computers. We tested the package on both intel- and motorola-powered machines to prove the usefulness of the package on the personal computer platforms. The package works better, as it is obvious, when there is more processing power and memory; another advantage of mainframe-sized machines lies in their batch execution facilities, useful for greater dimension irreps.

We think that the package can be used by the researchers in both experimental and theoretical physics as a tool to manage the irrep invariants and the products of irreps: for that use, with the processing of irreps of dimension less than 1000, the program gives answers nearly instantaneously, and the memory requirements are limited; this because the products take a time proportional to the highest dimension of the reduced irreps. As far as higher dimensional irreps products are needed, the time required grows rapidly after dimension 10000 irreps are reached.

Our necessities in supergravity theory required some products of 120 irreps of D5; so we tested the program in the worst case of the highest irrep in $120 \otimes 120$, the 4125, multiplied by another 120. First we obtained (on a Vax 4000, in batch)

$$120 \otimes 120 = 4125 \oplus 5940 \oplus 1050 \oplus \overline{1050} \oplus 945 \oplus 2 \cdot 210 \oplus 54 \oplus 45 \oplus 1 \quad (9)$$

in nearly 10 minutes of cpu time; then we did the products of each irrep with another 120:

$$\begin{aligned} 120 \otimes 4125 = {} & 70070 \oplus 192192 \oplus 48114 \oplus 34398 \oplus 48114 \oplus 27720 \\ & \oplus 36750 \oplus 2 \cdot 10560 \oplus 4312 \oplus 3696 \oplus \overline{3696} \oplus 2970 \oplus 1728 \oplus 120 \end{aligned} \quad (10)$$

in nearly 5 hours of cpu time.

This really displays the actual limit of our package: it will process irreps as long as the product does not exceed about dimension 1000000 (on our computing facilities), not only because of the time (which grows rapidly) but because of the memory requested to hold such huge weight systems in memory.

While we think that 10000×10000 dimension irrep products are really more than needed by the physics researchers, we report here some of the improvements we have in schedule to speed up the package and to reduce memory requirements.

- First the memory problem: here we have no great cure for this, due to the simple fact that *a dimension 1000000 irrep really means a weight system of nearly 100000 vectors*. Recalling that we have already reduced as possible the memory required by a `wplace` structure to:

  l bytes for the weight vector itself,

  1 short for degeneration,

  1 short for level (which must be made a long to permit irreps with level greater than 65000)

  which gives us the *minimum* of $l+4$ bytes per vector; hoping for as much degeneration as to reduce the need for weight vectors from 1000000 to 100000, we would have to find, for a 5-rank algebra, one megabyte of memory for a dimension 1000000 weight system. We really can not reduce the memory occupied by weight systems without introducing some sort of in-memory data compression, which will bring us, however, a speed penalty.

- Regarding the speed constraints, we have more directions, some of which are already in progress [8]:

  1. Use of the spindle shape property of weight systems. This optimization relies on the fact that the most part of time taken to construct weight systems is used in the lower half of the w.s., because of the recursive nature of much of the routines involved, such as `wtree` and (mostly) `freud` (this one is the principal cause of slowness, because it calls itself recursively from higher to lower levels.

The key point lies in the fact that we can truncate the elaboration at half the height of an irrep (which is known by `wheight`) and generate algorithmically the second half through a mirroring procedure which is specific to each algebra.

2. Creation of a *pointer cache* to facilitate the traversing of the w.s. list by routines such as `freud`; this is a simple and effective improvement, that will accelerate considerably the w.s. scanning routines. One has to enlarge the block definition in order to hold a set of #height pointers, each of which will point to the first `wplace` element at a each height.
3. A "technologic" improvement can be obtained by rewriting the routines that actually use `double` arguments into integer routines (the need for floating point arithmetics derives from the metric matrix, which is related to the inverse of the Cartan matrix).
4. A user-friendly improvement, useful for batch processing, would be a set of routines for managing lists of weight systems, such as to facilitate the construction of multiple products and products of sums of irreps; it will permit to treat sums of irreps as single entities, and we could directly compute products such as $(3 \oplus 6) \otimes (1 \oplus 8)$.

Acknowledgements

Thanks go to K. Lechner for his training on Lie algebra irrep techniques.

References

[1] W. Mc Kay, J. Patera and D. Sankoff, "Computers in Non Associative Rings and Algebras" eds R. Beck and B. Kolman (Academic Press, New York, 1977); J. Patera and D. Sankoff,"Tables of Branching Rules of Representations of Simple Algebras" (L'université de Montréal, 1973);

[2] W. Mc Kay and J. Patera, "Tables of Dimensions, Indices and Branching Rules for Representations of Simple Algebras" (Dekker, New York, 1981);

[3] R. N. Cahn, "Semi-Simple Lie Algebras and Their Representations" (Benjamin/Cummings, Menlo Park – California, 1984);

[4] S. B. Lippman, "C++ Primer" (Addison-Wesley, Reading – Massachusets, 1991);

[5] B. Stroustrup, "C++ Language" (Addison-Wesley, Reading – Massachusets, 1991);

[6] B. W. Kernighan and D. M. Ritchie, "The C Language" (Prentice-Hall, 1978);

[7] R. Slansky, *Phys. Rep.* **79**(1981)1;

[8] A. Candiello, *in preparation.*