WBase: a C package to reduce tensor products of Lie algebra representations. Description and new developments.*

Antonio Candiello
Dipartimento di Fisica, Università di Padova
Istituto Nazionale di Fisica Nucleare, Sezione di Padova
Padova, 35131, Italy

A non trivial application of a modern computer language (”C”) in a highly structured and object-oriented fashion is presented. The contest is that of Lie algebra representations (irreps), specifically the problem of reducing the products of irreps with the weight tree algorithm. The new `WBase 2.0` version with table-generation and Young tableaux display capabilities is introduced.

1. Introduction

Calculations in algebra representation theory, in particular decompositions of products of irreps, are needed in several sectors of physics. The Dynkin approach to the representation theory[1] is known by physicists thanks to its generality: all simple Lie algebras, included the exceptional ones, are described and manipulated in the same formal environment. In the Dynkin approach the algebras are described uniquely by the $l \times l$ Cartan matrix, where l is the rank of the algebra. The irreps of a given algebra are identified by a unique *highest weight vector* of l positive integers.

The purpouse of this contribution is to show the convenience of using modern computer programming techniques when applied to the Dynkin approach of algebra representation theory. Indeed, we have been able to construct a versatile algebra-manipulation package, named `WBase`,[2] such that: 1) `WBase` is a compact project, and the memory the Dynkin approach needs is used at best so that it works also on small computers; 2) `WBase` is easily upgradable with features.

With regard to the point 2) in this work we will describe the new features of the `WBase V2.0` version, a) the new table-generation routines; and b) the new Young-display support routines. The `WBase V2.0` now supports directly all Cartan algebras, both classical and exceptional.

Algorithms more specialized but faster than the Dynkin one use the extended Young diagrams. In `WBase V2.0` we added the extended Young diagram display capability for the classical algebras in order to analize these alternative methods.

*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.

2. Dynkin's approach to representation theory[1]

A simple Lie algebra in the Cartan-Weyl basis is described by a set of l simultaneously diagonalizable generators H_i and by the other generators E_α, satisfying

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

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

l is the *rank* of the algebra, and the set of l-vectors α_i are the *roots.* It results that all roots can be constructed via linear combinations by a set of l roots, called *simple roots.*

By the simple roots one then constructs the $l \times l$ *Cartan matrix*, which 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. In the Dynkin approach 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, takes care of generating algorithmically the related Cartan matrix (called `wcart` in `WBase`).

The *metric* G_{ij}, which is related to the inverse of the Cartan matrix (called `wmetr` in `WBase`) introduces a scalar product in the space of *weight vectors*, which are l-uples of integer numbers. Each irrep in an algebra is uniquely classified by a weight vector, the *highest weight* Λ whose components are all positive integers (the Dynkin labels). The different states in an irrep in a given irrep are again described by a weight vector w; the full set of all states of a given irrep is thus described by a set of weight vectors, called the *weight system*.

The dimension of an irrep Λ can be calculated with the help of the *Weyl formula* (encoded in the `weyl` function),

$$\dim(\Lambda) = \prod_{pos.roots \alpha} \frac{(\Lambda + \delta, \alpha)}{(\delta, \alpha)} \tag{3}$$

where Λ is the highest weight determining the irrep, $\delta = (1, \cdots, 1)$, and $(\ ,\ )$ is the scalar product constructed with the metric G_{ij}. The *positive roots* are the positive weight vectors of the weight system of the adjoint irrep.

The weight system of an irrep is needed to reduce the products of irreps, which is one of the main task of our package. The weight system is obtained by subtracting the simple roots α_i from the highest weight as described by the following recursive procedure (handled by `wtree`):

for all i *in* $1, \ldots, l$
    compute $u_k = \Lambda - k\alpha_i,\ k = 1, \cdots, u(i)$
    add all u_k *to the weight system*
    restart the procedure with $\Lambda := u_k$

The computational heaviest part of the construction of the weight system is the computation of the degeneration of each weight vector. It is computed by the *Freudenthal recursion formula* (encoded in `freud`), which needs the degeneration of previous levels,

$$\deg(w) = \frac{2\sum_{pos.roots\alpha}\sum_{k>0}\deg(w+k\alpha)\,(w+k\alpha,\alpha)}{\parallel \Lambda+\delta \parallel^2 - \parallel w+\delta \parallel^2} \tag{4}$$

with the initial condition $\deg(\Lambda) = 1$.

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 Λ_i in the equation

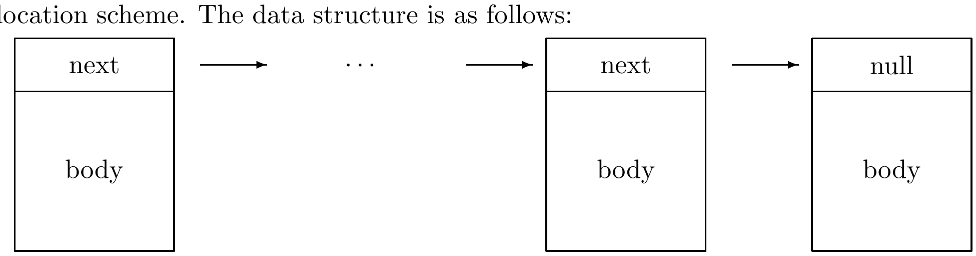
$$\Lambda_a \otimes \Lambda_b = \Lambda_1 \oplus \cdots \oplus \Lambda_n, \tag{5}$$

each with its degeneration, where each highest weights Λ identify uniquely an irrep. The algorithm (encoded in `wpdisp`) is the following. `wsyst`: generate the weight systems $ws(\Lambda_a) = \{w_{an}\}$, $ws(\Lambda_b) = \{w_{bm}\}$, and (`bprod`): construct the set $P = \{w_{an} + w_{bm}\}$; then, until P is empty, (`whighest`): find the highest weight Λ_i in P, (`wsyst`): generate the weight system $ws(\Lambda_i) = \{w_{ik}\}$, and (`bremove`): subtract from P each weight vector in $ws(\Lambda_i)$.

3. Dynamic data structures

The fundamental data type underlying our routines is the dynamically allocated (by `walloc`/`wfree`) weight vector `wvect`, with a run-time chosen length `wsize` $\equiv l$. The extended weight vector `wplace` contains also the degeneration and level informations of the weight vectors embedded in the weight systems. 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 for the weight systems.

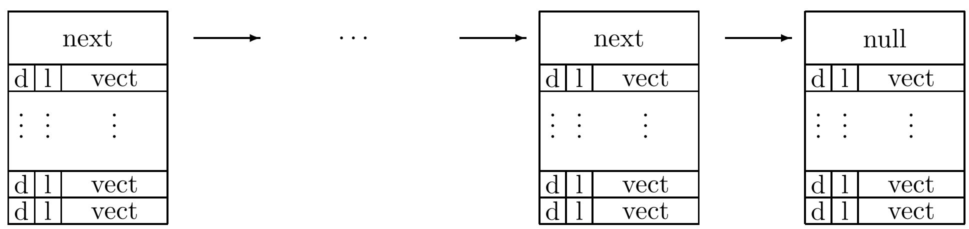
The difficulty in constructing such weight systems for arbitrary irreps is due to the fact that the dimension of the table needed to store the weight systems is not known in advance. The only solution that does not waste large amounts of memory and does not limit our routines more than the hardware does 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;
```

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 released with a call to `bfree`, linked blocks are released with a call to `bsfree` for 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

The task of setting up the algebra structures and handle the lists of vectors is taken by the following procedures:

`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()`. Multiple invocations of this pair of routines are needed for the construction of a table of several algebras.

`wread(hw)`, `wfdisp(hw)`, `wydisp(hw)`

These are the input/output routines. `wread` read a highest weight vector label from the standard input. `wfdisp(hw)` calls the functions `weyl`, `casimir` and `wheight` to display the related irrep informations on the standard output. `wydisp` is the new `WBase V2.0` extended Young tableaux display function available for all non exceptional algebra types. These routines need a valid `wvect` as handled by `walloc()/wfree()`.

`wsave(w,base,level)`, `wremove(w,base)`

Insert/remove the weight vector `w` in the list pointed by `base`. If `level` is greater than zero, `wsave` does a sorted insertion without degeneration increment, otherwise `w` is stored with its degeneration. `wremove` will remove a weight vector `w` from the list `base` only if the degeneration is just 1.

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

`wsyst` returns the full linked list of blocks of the weight system of highest weight `hw` along with the degenerations. It uses the recursive algorithm described in section 2, by calling `wsave` to store the weight vectors, then computes the degeneration with the Freudenthal formula. `bdisp(base)` displays to the standard output all entries of the weight system in `base`. The block list constructed by `wsyst` has to be deallocated with `bsfree`.

`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`.[a] The product routines are also available as iteration functions (see below).

User interface in `WBase V2.0`

In the file `wmain.c` it is implemented an ANSI C terminal-like interface with the user. A more sophisticated user interaction may be constructed taking this file as an example. Thanks to the new capabilities introduced in `WBase V2.0`, we had to add more options which are still one-letter options (for the details, refer directly to the source code).

5. Iterators

One of the standard conceptual device of the object-oriented technology is the *iterator*. The iterator hides the implementation details providing a consistent interface for moving through the data structure of the object considered. In `WBase` we introduced: 1) the iterator needed to scan the `wblock` list which contains the weight tree and 2) the iterator used to generate the `wblock` lists which contain the decomposed irreps of a product. New in `WBase V2.0` is 3) the table-generation iterator. Following is a short description of their use.

Scanning through the `wblock` *list*

It is done through the `pstart`/`pnext` iterators:

```
wplace *p;
pcurr pc;
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. 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`.

Getting the irreps of the decomposition of products

The iteration functions `wpstart(base)/wpnext(base,b)` that interface the construction of products return a `wblock` pointer to the full weight system of the reduced irrep. In order to display the weight tree of each irrep in the product of the two irreps with highest weight `hw1` and `hw2` the fragment of code is as follows:

```
wblock *base,*b;
base=bprod(wsyst(hw1),wsyst(hw2));
for(b=wpstart(base);b!=NULL;b=wpnext(b,base))
  bdisp(b);
```

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

Generating irreps of increasing dimensions

This is one of the main innovations of `WBase V2.0` that were only announced when we first introduced the package [2]. The table-generation procedure `wsequence` is used to produce increasing dimension irreps:

```
wblock *b=NULL;
wvect hw=walloc();
int dim;
while(b=wsequence(hw,b,maxdim,&dim))
 wfdisp(hw);
```

`wsequence` provides to: 1) allocate the first block when invoked with b=NULL the first time, 2) allocate eventual subsequent blocks, to store the encountered highest weights `hw`, and 3) deallocate all of them when `maxdim` is reached. In this last case it returns NULL. Like the other iteration procedures, `wsequence` has as an argument a specific iterator object (the `wblock` pointer `b`) which identify each irrep sequencing. It is then possible to nidificate multiple iterators to produce table of products.

In the next version of `WBase` we will probably transform the algebra initialization/destruction functions `wstartup/wcleanup` in iterators in order to extend the table-generation capabilities of `WBase V2.0` to multiple algebra tables. We will also probably switch to the C++ language to provide a more consistent iteration interface across the different data structures. The operator overloading capabilities of the C++ language will also simplify the interface of our package by unifying our list storage/remove functions (by overloading the `+=` and `-=` operators) and our input/output functions (by overloading the `<<` and `>>` operators).

6. Conclusions

The `WBase` package, in our opinion, represents a useful and self-contained demonstration of the convenience of new object-oriented software technology when combined with the C powerful dynamic data allocation facilities. As a by-product, we obtained what we think can be a useful tool for not too heavy irrep-related necessities of physicists. The Dynkin approach of Lie algebra representation theory helped us to maintain a unified and elegant structure to our package, however it must be noted that there are less general but faster algorithms[3] based on tensor and spinor manipulations useful in computing products of irreps.

References

1. R. Slansky, *Phys. Rep.* **79**, (1981) 1.
2. A. Candiello, *Comput. Phys. Comm.* **81**, (1994) 248.
3. G.R.E. Black, R.C. King and B.G. Wybourne, *J. Phys.* **A16**, (1983) 1555.
4. B. Stroustrup, *The C++ Programming Language*, 2nd Edition, (Addison-Wesley, Reading – Massachusets, 1991).