Finite dimensional Lie algebras

Anton Nazarov

March 16, 2010

Abstract

We present concise introduction to the representation theory of finite-dimensional Lie algebras and illustrate it with the computational algorithms implemented in Scheme.

1 Introduction

Representation theory of finite-dimensional Lie algebras is central to the study of continuous symmetries in physics. This theory is well-understood and there exist standard courses and textbooks on the subject [1], [humphreys]. Nevertheless some problems of the representation theory require extensive computation and no standard textbook on the computational algorithms is known to the author of this notes. There exists a volume [2], but it was written in 1970-es and have not been updated since, so its contents are limited to the early approaches and implementations on the old hardware which is unavailable now. Also some progress was made in the computational algorithms of the representation theory since the publication of [2]. It is important to mention series of papers by Patera et al. (see [3] and references therein) and books [4], [5] which introduce new and optimised algorithms although do not discuss the implementations. There exist several solid implementation of the core algorithms. We want to mention Maple package Coxeter/Weyl [6] and standalone programs LiE [7] and LambdaTensor [8]. These programs are solid and rather fast but have not seen any updates in last several years. Also they are not always convenient to use since they lack graphical user interface and interoperability with the popular programming languages and mathematical programs such as Mathematica, Python or Fortran. We want to summarise some basic notions and algorithms of representation theory in order to stimulate the emergence of more modern and universal software or at least give some tools to the scientists who by some reasons can not use the existing software.

Our implementation use programming language Scheme and is presented as the Literate program [9]. The choice of the language is due to high portability of its implementations [1], wide use of Scheme for the teaching [10] and personal preferences of the author.

[1] There exist Scheme implementations for UNIX, Windows, Linux, Mac OS, Palm OS, Windows CE/Pocket PC/Windows Mobile, Java platform and even micro-controllers, see [some website] for the details

2 Lie algebras

Definition 1. Lie algebra $\mathfrak{g}$ is a linear space with the bilinear operation

$$[\ ,\ ] : \mathfrak{g} \otimes \mathfrak{g} \to \mathfrak{g} \tag{1}$$

with the additional property that Jacoby identity holds

$$[x, [y, z]] + (\text{cyclic permutations}) = 0 \tag{2}$$

Lie algebras can be finite- or infinite-dimensional. Finite-dimensional Lie algebras are classified.

We will represent Lie algebra in the code as the object of class `lie-algebra`. We use very simple class system, which is described in the Appendix A.2. So `lie-algebra` is really an interface, but since our object system doesn't have interfaces we make it an abstract class and concrete classes such as `semisimple-lie-algebra` inherit it. Lie algebra is a vector space, so we really should create class for the vector space which contains methods for the change of basis. But for now we use only one standard basis for Lie algebras and `lie-algebra` is the subclass of the most genral class `object`.

2 ⟨*Lie algebra class .scm* 2⟩≡ (10c)

```
(class 'lie-algebra 'object
       ⟨Lie algebra methods .scm 3a⟩)
```

We need to add several definitions and describe a structure of Lie algebra in order to discuss fields and methods of `lie-algebra` class.

Definition 2. Ideal $\mathfrak{e} \subset \mathfrak{g}$: $[\mathfrak{g}, \mathfrak{e}] \subset \mathfrak{e}$

Definition 3. $\mathfrak{g}$ is a **simple Lie algebra** if $\mathfrak{g}$ has no proper ideal (except $\mathfrak{g}$ and $\{0\}$).

Definition 4. A Lie algebra $\mathfrak{g}$ is called **soluble** if $\mathfrak{g}^{(n)} = \{0\}$, where $\mathfrak{g}^{(n)} = [\mathfrak{g}^{(n-1)}, \mathfrak{g}^{(n-1)}]$.

Definition 5. $\mathfrak{g}$ is semisimple if its maximal soluble ideal $R \subset \mathfrak{g}$ is equal to $\{0\}$.

Semisimple Lie algebra is a direct sum of simple Lie algebras.

Since Lie algebra is a linear space, it can be specified by a set of generators $J_a \in \mathfrak{g}$ with the commutation relations $[J_a, J_b] = if^c_{ab}J_c$. f^c_{ab} are called **structure constants**. The number of generators is equal to the dimension of Lie algebra.

Now we need to discuss internal structre of Lie algebras, which can be representated with the code constructs and used for the study of algebra properties.

Definition 6. Cartan subalgebra $\mathfrak{h} \subset \mathfrak{g}$ - is the maximal commutative subalgebra of $\mathfrak{g}$. The dimension of the Cartan subalgebra $\mathfrak{h}$ is called **rank** of Lie algebra $\mathfrak{g}$.

3a ⟨*Lie algebra methods .scm* 3a⟩≡ (2)

```
'(rank ,(lambda (self)
              (error "Abstract class!")))
'(dimension ,(lambda (self)
              (error "Abstract class!")))
'(cartan-subalgebra ,(lambda (self)
              (error "Abstract class!")))
```

Commutative algebras constitute first concrete class of Lie algebras.

3b ⟨*Commutative algebra .scm* 3b⟩≡ (10c)

```
(class 'commutative-algebra 'lie-algebra
       '(dim 1)
       '(rank ,(lambda (self) (send self 'dim)))
       '(dimension ,(lambda (self) (send self 'dim)))
       '(cartan-subalgebra ,(lambda (self) self)))

;; The constructor
(define (make-commutative-algebra dim)
  (new 'commutative-algebra '(dim ,dim)))
```

TODO

It can be interesting and useful to add a method which show commutation relatons of the algebra in the symbolic form, but this task is left for the future

■

3 Semisimple Lie algebras

The structure of simple and semisimple Lie algebras can be encoded by the set of simple roots.

4a ⟨*Semisimple Lie algebra .scm* 4a⟩≡ (10c)

```
(class 'semisimple-lie-algebra 'lie-algebra
       '(simple-roots ())
       ⟨Semisimple Lie algebra methods .scm 4b⟩)
;; The constructor
(define (make-semisimple-lie-algebra simple-roots)
  (new 'semisimple-lie-algebra `(simple-roots ,simple-roots)))
```

3.1 Root systems

Definition 7. In the **Cartan-Weyl** basis the genrators are constructed as follows. First we choose the genrators $H_i \in \mathfrak{h}$ of the Cartan subalgebra. All these generators commute $[H_i, H_j] = 0$ and their representations can be diagonalised simultaneously. The remaining generators E_α are chosen to be a linear combinations of J_a such that

$$[H_i, E_\alpha] = \alpha_i E_\alpha \tag{3}$$

The vector $\alpha = (\alpha_1, \ldots, \alpha_r),\ r = rank(\mathfrak{g})$ is called a **root** and the corresponding operator E_α is called **ladder operator**.

The root α maps H_i to the number $\alpha(H_i) = \alpha_i$ hence $\alpha \in \mathfrak{h}^*$ - element of the dual to the Cartan subalgebra. The set of all the roos α is denoted by $\Delta = \{\alpha,\ \alpha$ - root$\}$. So it is natural for the `lie-algebra` class to have a method which lists all the roots.

4b ⟨*Semisimple Lie algebra methods .scm* 4b⟩≡ (4a)

```
`(rank ,(lambda (self)
    (length (send self 'simple-roots))))

`(cartan-subalgebra ,(lambda (self)
    (make-commutative-algebra (send self 'rank))))
⟨Root system methods .scm 6a⟩
```

All the roots are the linear combinations of simple roots with the integral coefficients. There exists discrete group of symmetry called **Weyl group** which can be used to produce all the roots from the set of simple roots.

Definition 8. Weyl group W of finite-dimension Lie algebra is a finite reflection group of $\mathfrak{h}^*$ generated by the basic reflections corresponding to the simple roots. Each simple root α_i corresponds to the reflection r_i in hyperplane orthogonal to α.

$$r_i\lambda = \lambda - \frac{(\lambda, \alpha_i)}{(\alpha_i, \alpha_i)}\alpha_i \tag{4}$$

For $w \in W$ there exists several equivalent realizations of the form $w = r_{i_1} \cdot r_{i_2} \dots r_{i_k},\ i_1 \dots i_k = 1..r$. The expression of the smallest *length* is called *reduced*.

We will implement represent Weyl group elements as the lists of numbers of basic reflection

$$w = r_{i_1} \cdot r_{i_2} \dots r_{i_k} \leftrightarrow (i_1, i_2, \dots, i_k) \tag{5}$$

Then we will implement action of Weyl group elements on $\mathfrak{h}^*$ using the recipies from [11], [12], but it will be done in the future. For now we are not implementing abstract construction of Weyl group since it is enough to be able to construct Weyl group orbits of $\mathfrak{h}^*$ elements.

Also sometimes we need to know the determinant of Weyl group element and we can take it into account using the following version of the `reflect` function, called `reflect-with-eps`.

5 ⟨*Functions for root systems .scm* 5⟩≡ (10c)

```
(define (co-root r)
  (div (mul 2 r) (prod r r)))

(define (reflect weight root)
  (sub weight (mul (prod weight (co-root root)) root)))

(define (reflect-with-eps weight root)
  (if (or (null? (cdr weight))
          (pair? (cdr weight)))
      (let ((v (reflect weight root)))
        (cons v (if (equal? weight v) 1 -1)))
      (let ((v (reflect (car weight) root)))
        (cons v (if (equal? (car weight) v) (cdr weight) (- (cdr weight)))))))

(define (eps weight)
  (if (or (null? (cdr weight))
          (pair? (cdr weight)))
      1
      (cdr weight)))

(define (no-eps weight)
  (if (or (null? (cdr weight))
          (pair? (cdr weight)))
      weight
      (car weight)))
```

Here *co-root* $\alpha^\vee$ is the element of the dual space $(\mathfrak{h}^*)^*$ which is identified with $\mathfrak{h}^* \approx \mathfrak{h}$ since we have scalar product on $\mathfrak{h}$.

Now we can implement wide range of root system related methods of `semisimple-lie-algebra` class

6a ⟨*Root system methods .scm* 6a⟩≡ (4b) 6b▷

```
'(simple-co-roots ,(lambda (self)
                     (map co-root (send self 'simple-roots))))
```

Definition 9. Cartan matrix with the elements

$$a_{ij} = (\alpha_i, \alpha_j^\vee) = \frac{2(\alpha_i, \alpha_j)}{(\alpha_j, \alpha_j)} \tag{6}$$

completely determines the set of simple roots and is useful for the classification and compact description of Lie algebras.

6b ⟨*Root system methods .scm* 6a⟩+≡ (4b) ◁6a 6c▷

```
'(cartan-matrix
  ,(lambda (self)
     (map (lambda (a)
            (map (lambda (av) (prod a av)) (send self 'simple-co-roots)))
          (send self 'simple-roots))))
```

We have already introduced Weyl group of reflections 8, now we implement procedure which constructs the union of Weyl group orbits of the set of weights.

6c ⟨*Root system methods .scm* 6a⟩+≡ (4b) ◁6b 7a▷

```
'(general-orbit , (lambda (self weights reflect)
    (let ((addon
           (filter (lambda (x) (not (element-of-set? x weights)))
                   (fold-right union-set '()
                               (map (lambda (w)
                                      (map
                                       (lambda (x) (reflect w x))
                                       (send self 'simple-roots)))
                                    weights)))))
      (if (null? addon)
          weights
          (send self 'general-orbit (union-set weights addon) reflect )))))

'(orbit ,
  (lambda (self weights)
    (send self 'general-orbit weights reflect)))

'(orbit-with-eps ,
  (lambda (self weights)
    (send self 'general-orbit weights reflect-with-eps)))
```

Definition 10. Fundamental weights form a basis $w_1, \ldots, w_r$ dual to the basis of simple co-roots $\alpha_1^\vee, \ldots, \alpha_r^\vee$.

7a ⟨*Root system methods .scm* 6a⟩+≡ (4b) ◁6c 7b▷

```
'(fundamental-weights
  ,(lambda (self)
     (map (lambda (x)
            (sum (map-n (lambda (y z) (mul y z)) x (send self 'simple-roots))))
          (matrix-inverse (send self 'cartan-matrix)))))
```

Weyl vector $\rho = \sum_i w_i = \frac{1}{2} \sum_{\alpha \in \Delta^+} \alpha$.

7b ⟨*Root system methods .scm* 6a⟩+≡ (4b) ◁7a 7c▷

```
'(rho ,(lambda (self)
         (sum (send self 'fundamental-weights))))
```

We can construct the full set of Lie algebra roots using Weyl symmetry.

7c ⟨*Root system methods .scm* 6a⟩+≡ (4b) ◁7b

```
'(roots ,(lambda (self)
           (send self 'orbit (send self 'simple-roots))))
```

3.2 Simple Lie algebras

Simple Lie algebras are classified by the use of Dynkin diagrams (which we will draw in the upcoming versions of this program) [1].

Here we explicitly construct simple roots for the classical series of finite-dimensional Lie algebras

8 ⟨*Simple Lie algebras .scm* 8⟩≡ (10c)

```
(define (make-simple-lie-algebra series rank)
  (define (simple-roots series rank)
    (cond ((eq? series 'A)
           (do ((i 0 (+ i 1))
                (base '()
                      (cons
                       (do ((j 0 (+ j 1))
                            (v '()
                               (cons
                                (cond ((= j i) 1)
                                      ((= j (+ i 1)) -1)
                                      (else 0))
                                v)))
                           ((> j rank) (reverse v)))
                       base)))
               ((= i rank) (reverse base))))
          ((eq? series 'B)
           (do ((i 0 (+ i 1))
                (base '()
                      (cons
                       (do ((j 0 (+ j 1))
                            (v '()
                               (cons
                                (cond ((= j i) 1)
                                      ((= j (+ i 1)) -1)
                                      (else 0))
                                v)))
                           ((= j rank) (reverse v)))
                       base)))
               ((= i rank) (reverse base))))
          ((eq? series 'C)
           (do ((i 0 (+ i 1))
                (base '()
                      (cons
                       (do ((j 0 (+ j 1))
                            (v '()
                               (cons
                                (cond ((and (= i (- rank 1)) (= i j)) 2)
                                      ((= j i) 1)
                                      ((= j (+ i 1)) -1)
                                      (else 0))
                                v)))
                           ((= j rank) (reverse v)))
```

```
                    base)))
                ((= i rank) (reverse base))))
            ((eq? series 'D)
             (do ((i 0 (+ i 1))
                  (base '()
                        (cons
                         (do ((j 0 (+ j 1))
                              (v '()
                                 (cons
                                  (cond ((and (= i (- rank 1)) (= (- i 1) j)) 1)
                                        ((= j i) 1)
                                        ((= j (+ i 1)) -1)
                                        (else 0))
                                  v)))
                             ((= j rank) (reverse v)))
                         base)))
                ((= i rank) (reverse base))))))
  (make-semisimple-lie-algebra (simple-roots series rank)))
```

4 Representations

Definition 11. Representation of Lie algebra $\mathfrak{g}$ on a linear space L is a homomorphism $V : \mathfrak{g} \to AutL$. It has the property

$$V([g,h])v = V(g)(V(h)v) - V(h)(V(g)v), \text{ for } g,h \in \mathfrak{g};\ v \in L \qquad (7)$$

There exists a classification of finite-dimensional representations of Lie algebras.

For the finite-dimensional representations we introduce abstract class `representation` and concrete class `highest-weight-representation`. Then we can describe tensor product of two highest weight representations as the object of class `representation` (but not of class `highest-weight-representation`.

9a ⟨*Representations .scm* 9a⟩≡ (10c) 9b▷

```
(class 'representation 'object
       `(dim ())
       `(multiplicity , (lambda (self weight)
                          (error "Class is abstract!")))
       '(lie-algebra ())
       )
```

9b ⟨*Representations .scm* 9a⟩+≡ (10c) ◁9a

```
(class 'highest-weight-representation 'representation
       '(highest-weight ())
       ⟨Anomalous weights computation .scm 10a⟩
       ⟨Weight multiplicities computation .scm 10b⟩)
```

We use the recurrent method of calculation of weight multiplicities described in [13], [14]. Weight multiplicities are determined using the set of anomalous points, which is constructed by the code below. We use the recurrent relation

$$m_{\xi}^{(\mu)} = -\sum_{w\in W\setminus e} \epsilon(w) m^{(\mu)}_{\xi-(w\cdot\rho-\rho)} + \sum_{w\in W} \epsilon(w)\delta_{(w(\mu+\rho)-\rho),\xi}. \qquad (8)$$

The second term on the right-hand side is constructed with the code:

10a ⟨*Anomalous weights computation .scm* 10a⟩≡ (9b)

```
'(anomalous-weights ,
  (lambda (self)
    (let* ((algebra (send self 'lie-algebra))
           (rho (send algebra 'rho)))
      (map (lambda (x)
             (sub x rho))
           (send algebra 'orbit (add (send self 'highest-weight) rho))))))
```

Then multiplicity of the weight ξ is calculated as the sum over the set $\omega\rho-\rho$, which we call "star"

10b ⟨*Weight multiplicities computation .scm* 10b⟩≡ (9b)

```
'(multiplicity , (lambda (self weight)
                   (let ((star ⟨Construct star .scm (never defined)⟩))
                     (-
                      (sum (map (lambda (wg) (* (eps wg) (delta weight wg)))))
                      (sum (map (lambda (wg)
                                  (* (eps wg)
                                     (send self 'multiplicity (sub weight wg))))
                                star))))))
```

5 General outline of the code

The code goes to several files of which `library.scm` is a compatibility layer which consists of auxiliary functions and allows portability between different Scheme implementations on different OS-es and hardware platforms. `library.scm` is described in the appendix A. All the mathematical code which is connected with Lie theory is collected in the file `lie-algebra.scm`.

10c ⟨*lie-algebra.scm* 10c⟩≡

⟨*Lie algebra class .scm* 2⟩
⟨*Commutative algebra .scm* 3b⟩
⟨*Semisimple Lie algebra .scm* 4a⟩
⟨*Functions for root systems .scm* 5⟩
⟨*Simple Lie algebras .scm* 8⟩
⟨*Representations .scm* 9a⟩

This code is written to file `lie-algebra.scm`.

A Library

Since one of our target implementations of Scheme is LispMe, which is only partially R4RS-compliant, we create compatibility layer of common functions. Also this library can be used to port the code for Lie algebras to other Scheme implementations.

11 ⟨*library.scm* 11⟩≡
⟨*Semi-standard Scheme procedures .scm* 12⟩
⟨*Sets .scm* 13⟩
⟨*Vector and matrix tools .scm* 16⟩
⟨*LispMe Objects .scm* 14⟩

This code is written to file `library.scm`.

A.1 Commonly used Scheme procedures

Due to the limitations of Palm OS platform functions such as `map` or `fold-left` in LispMe are in the separate library, which should be loaded by hand. Also function `map` accepts only 1-argument function and list, so we add these functions to our library.

12 ⟨*Semi-standard Scheme procedures .scm* 12⟩≡ (11)

```
(define nil ())
(define true #t)
(define false #f)

(define (zero? v)
        (= v 0))

(define (map op list)
        (if (null? list) ()
                (cons (op (car list)) (map op (cdr list)))))

(define (map-n op . lists)
    (define (map-n0 op  lists)
        (if (or (null? lists) (null? (car lists))) '()
        (cons (apply op (map car lists))
                (map-n0 op  (map cdr lists)))))
    (map-n0 op lists))

(define (accumulate op initial sequence)
  (if (null? sequence)
      initial
      (op (car sequence)
          (accumulate op initial (cdr sequence)))))

(define (filter predicate sequence)
  (cond ((null? sequence) ())
        ((predicate (car sequence))
         (cons (car sequence)
               (filter predicate (cdr sequence))))
        (else (filter predicate (cdr sequence)))))

(define (fold-left op initial sequence)
  (define (iter result rest)
    (if (null? rest)
        result
        (iter (op result (car rest))
              (cdr rest))))
  (iter initial sequence))

(define (fold-right op init s) (accumulate op init s))
```

We need couple of functions for sets. We represent set as list.

13 ⟨*Sets .scm* 13⟩≡ (11)

```
(define (element-of-set? x set)
  (cond ((null? set) false)
        ((equal? x (car set)) true)
        (else (element-of-set? x (cdr set)))))

(define (adjoin-set x set)
  (if (element-of-set? x set)
      set
      (cons x set)))

(define (intersection-set set1 set2)
  (cond ((or (null? set1) (null? set2)) '())
        ((element-of-set? (car set1) set2)
         (cons (car set1)
               (intersection-set (cdr set1) set2)))
        (else (intersection-set (cdr set1) set2))))

(define (union-set set1 set2)
        (if (null? set1) set2
                (union-set (cdr set1)
                           (adjoin-set (car set1) set2))
        )
)
```

A.2 Object system

It is natural to represent Lie algebras, root systems and representation as objects, so we need a class system. Since standard Scheme class systems such as TinyCLOS, GOOPS, SOS can not be used on LispMe due to the lack of hygienic macros, we use the minimal class system called LispMeObjects.

Here is the original documentation:

```
; Classes

 ; LispMeObjects
 ; http://c2.com/cgi/wiki?LispMeObjects
 ; written by Don Wells
 ; Create a new class with (class name super '(slot value)... '(method function)).
 ; Always use 'object as the super
 ; class at the very least.
 ; a function used as a method
 ; will take at least one argument
 ; self, the object that originally
 ; received the method.
 ; Invoke a function by sending the
 ; name and arguments to an
 ; object. (e.g. (send anObject 'add 'sum 10))
 ; where add is the method and sum and 10
 ; are arguments)
 ; Get the value of a slot by sending
 ; the slot's name.
 ; (e.g. (send anObject 'sum))
 ; Set the value of a slot by sending
 ; the set method defined on object.
 ; (e.g. (send anObject 'set 'sum 20))
 ; Always evaluate (clearClasses) before
 ; doing anything.

 ; an object is (superName (slotname value)... (methodName closure)...)
 ; a class is (className . object)
```

New classes are created with

```
(class '<classname> '<superclass> (list '<fieldname> <default-value>) ...)
```

Instances of classes:

```
(define <instance-name> (new '<classname> (list '<fieldname> <value>) ...))
```

14 ⟨*LispMe Objects .scm* 14⟩≡ (11)

```
(define *classes* '())
```

```
(define (clearClasses)
 (set! *classes*
   `((object #f
     (set ,setSlot)
     (super
        ,(lambda (self)
           (getClass (car self))))))))


(define (setSlot self aSlotName aValue)
 (let ((slot (assoc aSlotName (cdr self))))
   (cond
     ((not slot)
       (set-cdr! self
         (cons
           (list aSlotName  aValue)
           (cdr self))))
     (else
       (set-car! (cdr slot) aValue))))
 aValue)


(define (getClass aClass)
 (let ((class (assoc aClass *classes*)))
     (cond
       ((not class) #f)
       (else (cdr class)))))


(define (class aName aSuperName . aDefinition)
 (set! *classes*
   (cons
     (cons aName (cons aSuperName aDefinition))
     *classes*))
 aName)


(define (new aSuperName . args)
 (cons aSuperName args))


(define (send anObject aMessage . args)
 (sendWithSelf anObject anObject aMessage args))


(define (sendWithSelf self anObject aMessage args)
 (let
   ((superName (car anObject))
    (slot (assoc aMessage (cdr anObject))))
   (cond
```

```
      (slot (valueOfSlot self slot args))
      ((not superName) #f)
      (else
        (let ((superClass (getClass superName)))
          (cond
            ((not superClass) #f)
            (else
              (sendWithSelf self superClass aMessage args)))))))))


(define (valueOfSlot self theSlot args)
 (let ((value (cadr theSlot)))
   (cond
     ((procedure? value)
       (apply value (cons self args)))
     (else value))))
```

A.3 Vector and matrix manipulation

Here we have colleced simple tools for manipulation with vectors and matrices. They are important for the portability, since one of our target Scheme implementations doesn't support full R5RS-Scheme ([15]) and portable Scheme libraries such as SLIB.

16 ⟨*Vector and matrix tools .scm* 16⟩≡ (11)
⟨*Vector operations .scm* 17⟩
⟨*Matrix inverse .scm* 18a⟩

We represent vectors with lists since for now we don't want to have object system A.2 overhead. It can be changed in the future.

17 ⟨*Vector operations .scm* 17⟩≡ (16)

```
(define (add a b)
  (map-n + a b))

(define (sum list)
  (fold-left add (car list) (cdr list)))

(define (sub a b)
  (map-n - a b))

(define (mul a b)
  (cond ((and (list? a) (number? b)) (map (lambda (x) (* x b)) a))
        ((and (number? a) (list? b)) (mul b a))
        ((and (number? a) (number? b)) (* a b))
        (else (error "Can not multiply"))))

(define (div a b)
  (map (lambda (x) (/ x b)) a))

(define (prod a b)
  (fold-left + 0 (map-n * a b)))
```

For the matrix inverse and determinant we use the implementation with the cofactors extracted from the SLIB. It is not the fastest and should be replaced with the Gaussian elimination.

18a ⟨*Matrix inverse .scm* 18a⟩≡ (16)

```
(define (matrix-cofactor mat i j)
  (define (butnth n lst)
    (if (<= n 1)
        (cdr lst)
        (cons (car lst)
              (butnth (+ -1 n) (cdr lst)))))
  (define (minor matrix i j)
    (map (lambda (x)
           (butnth j x))
         (butnth i mat)))
  (* (if (odd? (+ i j)) -1 1)
     (matrix-determinant (minor mat i j))))

(define (matrix-determinant mat)
  (let ((n (length mat)))
    (if (eqv? 1 n) (caar mat)
        (do ((j n (+ -1 j))
             (ans 0 (+ ans (* (list-ref (car mat) (+ -1 j))
                              (matrix-cofactor mat 1 j)))))
            ((<= j 0) ans)))))


(define (matrix-inverse mat)
  (let* ((det (matrix-determinant mat))
         (rank (length mat)))
    (and (not (zero? det))
         (do ((i rank (+ -1 i))
              (inv '() (cons
                        (do ((j rank (+ -1 j))
                             (row '()
                                  (cons (/ (matrix-cofactor mat j i) det) row)))
                            ((<= j 0) row))
                        inv)))
             ((<= i 0) inv)))))
```

A.4 Supplementary mathematical functions

18b ⟨*Supplementary math .scm* 18b⟩≡

```
(define (delta weight1 weight2)
  (equal? (no-eps weight1) (no-eps weight2)))
```

References

[1] R. Carter, *Lie algebras of finite and affine type*. Cambridge University Press, 2005.

[2] J. Belinfante and B. Kolman, *A survey of Lie groups and Lie algebras with applications and computational methods*. Society for Industrial Mathematics, 1989.

[3] R. Moody and J. Patera, "Fast recursion formula for weight multiplicities," *AMERICAN MATHEMATICAL SOCIETY* **7** (1982) no. 1, .

[4] S. Kass, R. Moody, J. Patera, and R. Slansky, *Affine Lie algebras, weight multiplicities, and branching rules*. Sl, 1990.

[5] M. Bremner, R. Moody, and J. Patera, *Tables of dominant weight multiplicities for representations of simple Lie algebras*. Dekker, 1985.

[6] J. Stembridge, "A Maple package for symmetric functions," *Journal of Symbolic Computation* **20** (1995) no. 5-6, 755–758. `http://www.math.lsa.umich.edu/~jrs/maple.html`.

[7] M. Van Leeuwen, "LiE, a software package for Lie group computations," *Euromath Bull* **1** (1994) no. 2, 83–94. `http://www-math.univ-poitiers.fr/~maavl/LiE/`.

[8] T. Fischbacher, "Introducing LambdaTensor1. 0-A package for explicit symbolic and numeric Lie algebra and Lie group calculations," *Arxiv preprint hep-th/0208218* (2002) .

[9] D. Knuth, "Literate programming,".

[10] H. Abelson, G. Sussman, and J. Sussman, *Structure and interpretation of computer programs*. Citeseer, 1996.

[11] J. Stembridge, "Computational aspects of root systems, Coxeter groups, and Weyl characters, Interaction of combinatorics and representation theory, MSJ Mem., vol. 11," *Math. Soc. Japan, Tokyo* (2001) 1–38.

[12] W. Casselman, "Machine calculations in Weyl groups," *Inventiones mathematicae* **116** (1994) no. 1, 95–108.

[13] V. Lyakhovsky, S. Melnikov, *et al.*, "Recursion relations and branching rules for simple Lie algebras," *Journal of Physics A-Mathematical and General* **29** (1996) no. 5, 1075–1088, `q-alg/9505006`.

[14] P. Kulish and V. Lyakhovsky, "String Functions for Affine Lie Algebras Integrable Modules," *Symmetry, Integrability and Geometry: Methods and Applications* **4** , `arXiv:0812.2381 [math.RT]`.

[15] R. Kelsey, W. Clinger, J. Rees, H. Abelson, N. Adams IV, D. Bartley, G. Brooks, R. Dybvig, D. Friedman, R. Halstead, *et al.*, "Revisedˆ 5 Report on the Algorithmic Language Scheme,".