Finite dimensional Lie algebras

Anton Nazarov

November 27, 2009

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 [Belinfante], 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 [Belinfante]. It is important to mention series of papers by Patera et al. (see [2] and references therein) and books [3], [Kass-Moody-Slansky on finite dimensional algebras] which introduce new and optimised algorithms although do not discuss the implementation. There exist several solid implementation of the core algorithms. We want to mention Maple package Coxeter/Weyl [4] and standalone programs LiE [5] and LambdaTensor [6]. 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 [Knuth]. The choice of the language is due to high portability of its implementations [1], wide use of Scheme for the teaching [SICP] 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 Definitions

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.

Definition 2. Representation of Lie algebra $\mathfrak{g}$ on a linear space L is a homomorphism $V : \mathfrak{g} \to AutL$. It has the property

$$V([g, h])v = V(g)(V(h)v) - V(h)(V(g)v), \text{ for } g, h \in \mathfrak{g};\ v \in L \tag{3}$$

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

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 3. Cartan subalgebra $\mathfrak{h} \subset \mathfrak{g}$ - is the maximal commutative subalgebra of $\mathfrak{g}$.

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.

2 ⟨*library.scm* 2⟩≡
⟨*Semi-standard Scheme procedures .scm* 3⟩
⟨*Sets .scm* 4⟩
⟨*LispMe Objects .scm* 5⟩
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.

3 ⟨*Semi-standard Scheme procedures .scm* 3⟩≡ (2)

```
(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.

4 ⟨*Sets .scm* 4⟩≡ (2)

```
(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))
        )
)
```

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>) ...))
```

5 ⟨*LispMe Objects .scm* 5⟩≡ (2)

```
  (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))))
```

References

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

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

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

[4] 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`.

[5] 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/`.

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