Euromath Bulletin
Vol. 1, No. 2, 1994

LiE, A software package for Lie group computations

M. A. A. van Leeuwen
CWI
Postbus 4079
1009 AB Amsterdam
The Netherlands

Abstract

A description is given of LiE, a specialised computer algebra package for computations concerning Lie groups and algebras, and their representations. The functionality of the package is demonstrated by sample computations, and the structure of the program and the algorithms implemented in it are discussed.

1 Introduction

LiE is a computer algebra package for computations with Lie groups and their representations, which has been developed at CWI, Amsterdam. The package offers an extensive library of functions implementing fundamental algorithms related to the theory of reductive Lie groups, in the following areas: root systems, the Weyl group and its action on the root and weight lattices, semisimple elements and their centralisers, highest weight modules, finite dimensional representations, characters, decompositions of tensor products, restrictions to subgroups (branching), symmetric group characters, symmetric and alternating tensor powers, and more general plethysms. The functions are available within an interactive programming environment, which incorporates a programming language that provides variables, control structures and user–defined functions. This allows easy and flexible handling of input and output data, and enables customisation of the package by the user to particular applications. A collection of example programs is supplied, showing how LiE can be applied to various kinds of problems. LiE comes with an extensive manual [Lee] (a hard–cover book of over 100 pages) which not only gives an introduction to the use of LiE, but also background information about the mathematical field covered by the package, full documentation on the programming language as well as on all functions in the library, and a discussion of the example programs. In addition to this LiE provides on–line help on all its features, as well as theoretical information about Lie groups and related topics.

2 A guided tour

We will illustrate the possibilities and limitations of the package by running an example session. We shall assume that the reader is familiar with concepts from the theory of Lie groups (or algebras) and their representations, and which can be found for instance in [Hum]; the main purpose of the examples is to show what kind of questions arising in that theory can be handled computationally using LiE, and in what manner.

After starting LiE the following is printed on the terminal screen.

```
$ lie

LiE version 2.0 created at Tue May 26 10:53:37 MET DST
1992
Authors: Arjeh M. Cohen, Marc van Leeuwen, Bert Lisser.

type '?help' for help information
type '?' for a list of help entries.
>
```

Had this not been a guided tour, then it would at least have been immediately clear how to get some help. The '>' sign at the end is the prompt of LiE telling us that it is ready to accept commands; henceforth we shall print this prompt only before each line of user input, distinguishing it from lines of output. As this is our first encounter, let us start with

```
> Hello
Hello is not defined.
```

So LiE does not return our greeting, but still we can learn something from this. First, typing <Return> was sufficient to get LiE's attention (by contrast some programs like *Maple* remain passive until a semicolon is entered). More importantly this shows that LiE is not a truly symbolic package, since most of such packages would accept 'Hello' as being a formal symbol, and with no further evaluation possible, return it unchanged as answer[1]. On the other hand, like almost all Computer Algebra programs, you can use LiE for doing ordinary arithmetic.

```
> 7*(3^2+2^2)
    91
> 1993/25
    79
> 31^23
    2001331164404928026413872424429539 1
```

As one sees, LiE only calculates with integral numbers (whence the remainder of the division is dropped), and (practically speaking) there is no bound on the magnitude of these numbers. LiE also has vector and matrices with integral entries

```
> [3, -5, 21-13]
    [3,-5,8]
>  [[1,2,3],$,[4,5,6]]
    [
    [1, 2,3],
    [3,-5,8],
```

1 In addition, AXIOM would try to clarify its response by adding: "Type: Variable Hello".

```
      [4,  5,6]
      ]
> m=$
> m[2,3]
      8
> m[1]
      [1,2,3]
```

The special symbol '$' stands for the previous result, and results can also be stored more permanently using variables, as the example shows. Besides stacking vectors of equal length to form the rows of a matrix, as is done above, we may also combine them formally, possibly with integer multiplicities (thus forming a multiset of vectors, except that the multiplicities might be negative) into what is called a polynomial in LiE. To this end we prefix each constituent with an 'X' and possibly with a multiplicity, and add everything together.

```
>  3X[4,-1,0]+X[3,3,2]-5X  m[3]
     1X[3,  3,2]  +  3X[4,-1,0]  -  5X[4,  5,6]
```

As the notation suggests, polynomials are considered as a sum of terms, and arithmetic operations are accordingly defined for polynomials, where individual terms are multiplied by adding their vectors (''exponents'') and multiplying their coefficients.

```
>  $*(-X[0,2,1]+3X[1,-2,-1])
    -1X[3,  5,  3]  +  5X[4,  7,  7]  +  9X[5,-3,-1]  -  15X[5,  3,  5]
```

This completes the list of the main data types used in LiE, except for the type used to indicate the type of Lie group for which computations are desired. To this end the standard classification of simple Lie groups is used, so for instance 'A2' and 'F4' denote simple groups of types A_2 and F_4 respectively (the first of which is also known as **SL**(3, **C**)); to resolve a possible ambiguity, only simply connected simple groups are considered (since they admit the most representations), whence 'A2' does not denote **PSL**(3, **C**). Semisimple groups may be formed by concatenation (e.g., 'B4B4E7') and reductive groups by adding an n–dimensional complex torus Tn. As a trivial example of a library function taking a group as parameter, we can request the Dynkin diagram to be drawn:

```
>  diagram(E6B3T2)

          O  2
          |
          |
O---O---O---O---O
1   3   4   5   6
E6

O---O=>=O
7   8   9
B3

With  2-dimensional  central  torus.
```

The most important aspect of this function is that it provides the standard numbering of the fundamental roots associated with the node of the Dynkin diagram (the program follows Bourbaki's labeling). As a matter of fact, nearly all library functions take a group as final parameter, and since most of the time this is likely to be the same group, a defaulting mechanism

for these parameters is provided. As group to work in we choose the simple group of type D_4, i.e., **Spin**(8, **C**) which covers **SO**(8, **C**), and we display its diagram:

```
> setdefault D4
> diagram

    O 3
    |
    |
O---O---O
1   2   4
D4
```

First we ask its dimension, the set of positive roots (each expressed as integer vector on the basis of fundamental roots) and the order of its Weyl group

```
> dim
     28
> pos_roots
     [
     [1,0,0,0],
     [0,1,0,0],
     [0,0,1,0],
     [0,0,0,1],
     [1,1,0,0],
     [0,1,1,0],
     [0,1,0,1],
     [1,1,1,0],
     [1,1,0,1],
     [0,1,1,1],
     [1,1,1,1],
     [1,2,1,1]
     ]
> W_order
     192
```

Each pair of opposite roots $\{\alpha, -\alpha\}$ is the root system of a subgroup of type A_1, whose maximal torus is a one–dimensional subtorus of the maximal torus T of the full group; we will now determine the roots of the centraliser of this subtorus, taking for α the third fundamental root $\alpha_3 = [0, 0, 1, 0]$. The subtorus can then be expressed as $\{\exp(t\, h_\alpha) \mid t \in \mathbf{C}\}$ where h_α is the Lie algebra element which is the coroot of α; expressed on the basis of fundamental coroots, h_α is again $[0, 0, 1, 0]$. The function *cent_roots* which we shall use can compute centralisers of elements of the form $\exp(\frac{2\pi i}{n} h_\alpha)$ as well as those of entire subtori; to indicate that we want the latter we append a 0 to the vector expressing h_α; we also compute the type of the centraliser subgroup.

```
>  cent_roots([0,0,1,0,0])
     [
     [1,0,0,0],
     [0,0,0,1],
     [1,2,1,1]
```

```
      ]
> Cartan_type($)
    A1A1A1T1
```

So the first and fourth fundamental roots are in the centraliser (indeed their nodes are not connected with the third in the diagram), as well as the highest root. If we make the similar computation for the central element $\exp(\frac{2\pi i}{2} h_\alpha)$ of the type A_1 subgroup instead of the whole subtorus, we also get the root α itself in the centraliser:

```
> cent_roots([0,0,1,0,2])
    [
    [1,0,0,0],
    [0,0,1,0],
    [0,0,0,1],
    [1,2,1,1]
    ]
> Cartan_type($)
    A1A1A1A1
```

We can also ask how this toral element acts on any representation of D_4 by means of the function *spectrum*. An element of finite order n will have eigenvalues of the form $e^{2k\pi i/n}$ only, and *spectrum* records the multiplicity of this eigenvalue as the coefficient of $X^{[k]}$ in a polynomial. Doing the computation for the adjoint representation confirms the centraliser information just computed.

```
> spectrum(adjoint,[0,0,1,0,2])
    12X[0]  +16X[1]
> dim(A1A1A1A1)
    12
```

So this element acts with eigenvalue 1 precisely on the Lie algebra of its centraliser, and with eigenvalue -1 on a 16–dimensional complement of this subalgebra within the full Lie algebra.

To work with arbitrary weights rather than just roots we need to use the basis of fundamental weights instead of the fundamental roots. The Cartan matrix produced by the function *Cartan* expresses the fundamental roots on the weight basis, and matrix multiplication by it can be used for coordinate transformation; for instance to express all positive roots on the weight basis we can say

```
> pos_roots*Cartan
    [
    [ 2,-1, 0, 0],
    [-1, 2,-1,-1],
    [ 0,-1, 2, 0],
    [ 0,-1, 0, 2],
    [ 1, 1,-1,-1],
    [-1, 1, 1,-1],
    [-1, 1,-1, 1],
    [ 1, 0, 1,-1],
    [ 1, 0,-1, 1],
    [-1, 0, 1, 1],
    [ 1,-1, 1, 1],
    [ 0, 1, 0, 0]
    ]
```

The fact that each row of this result except the last has at least one negative entry reflects the fact that among the positive roots only the highest root is dominant. LiE also knows about the Weyl group and how it acts on the weight lattice. For instance, putting $v=[2,-1,0,0]$ and calling *W_orbit(v)* produces a 24–row matrix whose rows consist of those of the previous result together with their negatives. We can also find the dominant weight in the orbit of v directly, and a Weyl word (a product of simple reflections in the Weyl group) that will send v to this dominant representative.

```
> v=[2,-1,0,0]
> dominant(v)
    [0,1,0,0]
> W_word(v)
    [2,3,4,2]
> W_action(v,$)
    [0,1,0,0]
```

Next let us find the matrix for the reflection in the before–last positive root on the weight lattice, express it as a Weyl word, and check that its square equals the identity element in W.

```
> reflection(pos_roots[11])
    [
    [  0,1,-1,-1],
    [-1,2,-1,-1],
    [-1,1,  0,-1],
    [-1,1,-1,  0]
    ]
> W_word($)
    [3,1,2,4,2,3,1]
> $^$
    [3,1,2,4,2,3,1,3,1,2,4,2,3,1]
> reduce($)
    []
```

Apart from reducing Weyl words to minimal expressions, we can also obtain a *canonical* reduced expression, namely the lexicographically first reduced expression:

```
> canonical([4,3,1,2,3,4])
    [1,3,2,4,2,3]
```

LiE provides many other useful functions for dealing with Weyl groups, amongst other for computing the Bruhat order and Kazhdan–Lusztig polynomials; for instance

```
> KL_poly([3],[3,2,1,4,2,3])
  1X[2]  +2X[1]  +1X[0]
```

The most powerful algorithms of LiE are those that deal with representations of Lie groups. Irreducible representations can be specified by their highest weight, and by combining several dominant weights into a polynomial we may also describe reducible representations; such a polynomial is called a decomposition polynomial. One important quantity attached to such a representation is its character, i.e., the set of all occurring weights with their respective multiplicities; the polynomial describing this multiset is called the character polynomial of the representation. Since characters are usually quite large and also symmetric under the Weyl group, it is sufficient and more efficient to record only the dominant terms in the character polynomial, and it is this dominant character polynomial that is computed by the function *dom_char*. For the irreducible representation with highest weight $[2,1,1,0]$ we

now compute the dimension, the dominant character, the number of terms in the dominant and full character, and we check that the coefficients of character polynomial add up to the dimension.

```
> dim([2,1,1,0])
    2800
> dom_char([2,1,1,0])
   33X[0,0,1,0] + 6X[0,0,1,2] + 3X[0,0,3,0] +12X[0,1,1,0] +
    1X[0,2,1,0] +21X[1,0,0,1] + 2X[1,0,2,1] + 4X[1,1,0,1] +
    9X[2,0,1,0] + 1X[2,1,1,0] + 2X[3,0,0,1]
> [ length($), length(W_orbit($)), dim(W_orbit($),T4) ]
    [11,528,2800]
```

Note that for the last computation we specified that the character polynomial should be considered as a decomposition polynomial for the maximal torus T_4, thereby overriding the default group D_4 as argument to *dim*. LiE has various ways of computing representations from other ones, in which case they are always described by their decomposition polynomials (which is the most economic form); an important example is formation of tensor products. Here we compute the tensor product of the representation above with the adjoint representation; we also check the dimension, which should be 28 times (namely $dim(D_4)$) the dimension of the original representation.

```
> tensor(X[2,1,1,0],adjoint)
 1X[0,2,1,0] +1X[1,0,2,1] +1X[1,1,0,1] +1X[1,1,2,1] +1X[1,2,0,1] +
 1X[2,0,1,0] +1X[2,0,1,2] +1X[2,0,3,0] +3X[2,1,1,0] +1X[2,2,1,0] +
 1X[3,0,0,1] +1X[3,0,2,1] +1X[3,1,0,1] +1X[4,0,1,0]
> [ 28*dim([2,1,1,0]), dim($) ]
    [78400,78400]
```

Another important operation is branching, i.e., viewing a representation of a group as a representation of another group via a group morphism, usually the embedding of a subgroup. To this end LiE requires a description of the group morphism, in the form of a matrix describing to which weights for the subgroup the fundamental weights of the original group correspond. If the subgroup is given by a subset of the root system, then LiE can compute this restriction matrix for you. Suppose for instance we are interested in the subgroup with roots [0, 0, 0, 1], [0, 1, 1, 1] and [1, 2, 1, 1]. We first compute a basis for the subset of roots spanned by these, then the type of the subsystem and its restriction matrix, and finally proceed to compute the restriction of the irreducible representation considered before.

```
> closure([[0,0,0,1], [0,1,1,1], [1,2,1,1]])
    [
    [0,1,1,0],
    [0,0,0,1],
    [1,1,0,0]
    ]
> h=Cartan_type($); r=res_mat($); print(r); h
    [
    [0,0,1,1],
    [1,0,1,0],
    [1,0,0,1],
    [0,1,0,0]
    ]
    A3T1
> branch([2,1,1,0],h,r)
```

```
     1X[0,0,3, 1] +1X[0,1,1, 1] +1X[0,1,3,-1] +1X[0,1,3, 3] +
     1X[0,2,1,-1] +1X[0,2,1, 3] +1X[1,0,2,-1] +1X[1,0,2, 3] +
     1X[1,0,4, 1] +1X[1,1,0,-1] +1X[1,1,2,-3] +2X[1,1,2, 1] +
     1X[1,1,2, 5] +1X[1,2,0,-3] +1X[1,2,0, 1] +1X[2,0,1,-3] +
     1X[2,0,1, 1] +1X[2,0,3,-1] +1X[2,0,3, 3] +1X[2,1,1,-5] +
     2X[2,1,1,-1] +1X[2,1,1, 3] +1X[3,0,0,-1] +1X[3,0,2,-3] +
     1X[3,0,2, 1] +1X[3,1,0,-3] +1X[3,1,0, 1] +1X[4,0,1,-1]
> dim($,A3T1)
    2800
```

This computation concludes our tour. There are many more functions that have not been discussed (such as computing symmetric group characters, symmetric and alternating tensor powers of representations, and plethyms) but at least we have seen examples of functions in all the main areas covered by the library of LiE.

3 The structure of LiE

LiE consists of a library of built–in functions, a user programming language which makes the functions available, and some other components such as a help system. We will now briefly discuss these components and how they fit together.

3.1 The library of algorithms

We have seen many examples of functions that are built into LiE. All these are written in the implementation language of LiE, which is 'C', and compiled into the program, and this library forms a major part of the program (about half of all the code). The basic core of LiE provides general support routines, such as memory management and support of the basic data types (e.g., integer and polynomial arithmetic); the specific Lie group algorithms are implemented as ordinary functions in C based on these basic facilities. This approach was chosen because it enables high efficiency (certainly compared with algorithms written in an interpreted language, such as that of many Computer Algebra systems) which is of vital importance for practical applicability of many of the algorithms involved, while most algorithms and the data types they use are sufficiently straightforward that an ordinary programming language can be used without great difficulty.

To give an idea of the efficiency considerations involved, certain basic algorithms such as character computation have also been implemented in *Maple*, and were found to run at least 10 times, but often about 100 times slower than the LiE routines. More concretely, the examples above, which involve a modest size group and modest weights, deliver their answers almost instantaneously on a fast workstation, and at most within seconds (for the tensoring and branching) on my slow and outdated home computer; had they been computed in a general purpose system the more complex computations would probably have taken minutes even on a fast machine (we can't be sure since we know of no package in any such system implementing the more complex algorithms).

3.1.1 Root systems and the Weyl group

We mention the particular algorithms used for the most crucial computations. In most cases the group dependent parts are based directly on the Cartan matrix, which is computed once and stored for each simple group used. The system of positive roots is generated straightforwardly from the fundamental roots and also stored since they are needed for instance for character computations. For working with subsystems of roots some original algorithms were developed, for instance for *closure*, which takes a set of roots spanning a subsystem and repeatedly makes it a "more fundamental" system by adjusting individual pairs of roots.

The action of simple reflections on either the root or weight lattice can be easily expressed in terms of the Cartan matrix; however only a few entries of root or weight vectors are affected by any one reflection, due to the sparseness of the Cartan matrix, and special code is used to exploit this fact. Since simple reflections are so easily performed and it is trivial to find out for any weight on which side of the hyperplane of any fundamental reflection it lies, most Weyl group operations are computed internally with the help of the action on weight vectors. This has an additional advantage over the use of Weyl words that the size of the data (the image of a single vector suffices) does not vary with the Weyl group element.

3.1.2 Highest weight representations

Dimensions of irreducible representations are computed by Weyl's dimension formula, which is astonishingly efficient: it is almost the only operation you can expect to be able to perform for arbitrarily high weights. Dominant characters are computed in two steps: first the set of dominant weights occurring is determined by a process of repeated subtraction of positive roots, then the individual multiplicities of these weights are determined by Freudenthal's recursion formula (working from the highest weight downwards). This method is much more efficient (especially for large groups) than Weyl's character formula, which involves a summation over the Weyl group. (This incidentally shows that asymptotic analysis can be very deceptive, since it tells you that Weyl's formula must be more efficient in the limit of high weights. The reason is that it involves only a constant number of arithmetic operations for each multiplicity computed, whereas Freudenthal's formula involves increasingly many previous results the further down one gets; the mentioned constant however involves the order of the Weyl group, which for E_8 for instance is 696729600.)

3.1.3 Weyl orbit traversal

Although iteration over the Weyl group is generally to be avoided due to its potentially immense size, we know no general algorithms for tensor product decomposition and branching that do not at least involve traversal of Weyl group orbits. In any case it is vital to use a method that exploits the stabiliser subgroup of the weight if non–trivial, since most small weights for large groups do in fact have substantial stabilisers and therefore orbits of acceptable size. Also one should avoid storing the whole orbit at once since cancellations in the result are common, and contrary to time, memory is fundamentally bounded.

In LiE a general method for traversing Weyl orbits is implemented which satisfies these requirements, and which has as fundamental ingredient a routine that generates all distinct permutations of a sequence of not necessarily distinct numbers. For groups of classical type this routine, or a simple generalisation of it, suffices, after applying a coordinate transformation to the so–called ε–basis. For the exceptional groups the largest subgroup of classical type is used, and the cosets for this subgroup are generated by a method which uses a Coxeter element in the Weyl group, but is also partially *ad hoc*; the classical subgroups used are A_2 in G_2 (index 2), B_4 in F_4 (index 3), D_5 in E_6 (index 27), A_7 in E_7 (index 72), and D_8 in E_8 (index 135).

3.1.4 Tensor products and branching

For tensor product computations Klymik's formula is used, generating the full character only for the smaller factor as determined by dimension comparison. For the special case of groups of type A_n however, a fast algorithm based on the Littlewood–Richardson rule is implemented as an alternative means to compute tensor product decompositions; it is especially useful when the rank of the group is high. For branching, the full character of the representation is generated and transformed by the restriction matrix; any weights which are dominant after restriction are collected and the result is decomposed by repeated subtraction of the dominant character of the highest remaining weight.

3.1.5 Weight collection

One interesting experience we had during the construction of LiE is the dramatic impact that improvements to rather mundane routines can have, such as the one for collecting weights with multiplicities into a polynomial, as they are produced by various algorithms. For instance, we originally used a quicksort routine for ordering the terms of a polynomial, but this routine far from fulfilled its pretentious name, since in practice its argument was often already almost sorted, making quicksort quite slow; this problem was remedied by using a heapsort routine instead.

Even more important was the decision when to sort: sorting frequently is quite inefficient, whereas gathering all terms before sorting may require an unacceptably large amount of memory. The method implemented in LiE is a hybrid one which uses both a sorted and an unsorted accumulator: every new term is quickly looked up in the sorted part to see if it can be incorporated by simply adjusting a coefficient; if not, it is added at the end of the unsorted part. Whenever the unsorted part grows to exceed the sorted part in size, it is sorted and merged to form a new sorted part.

3.2 The language of LiE

In the examples above we mainly used the interactive interface of LiE to call various built–in routines and pass results from one as input to another. In fact this uses only a small part of the language understood by the interpreter, which is a full–fledged programming language with variables, control structures and user defined functions. Now, as the reader who is familiar with other Computer Algebra packages will probably know, these usually proudly present their own unique idiosyncratic programming language; one might expect that the same holds for LiE. In fact however, although the language supported by LiE is certainly unique (this could hardly have been avoided), its structure is fairly regular, and it certainly takes no pride in any peculiarities that may still be present. It must however be admitted that the language has a rather limited scope, and is not well suited for the formulation of many algorithms in areas remote from that where the built–in functions operate; this is mostly due to the very restricted set of data types currently supported.

The language is a pretty straightforward procedural language, with ordinary programming variables (there can be no confusion with symbolic variables because of the absence of the latter), which can assume values of any of the six types available: integer, vector, matrix, polynomial, group and text, and which are always fully initialised. The repertoire of control structures is similar to that of Pascal or 'C', and there are functions with strict call–by–value

semantics. Importantly, values of composite types are handled with the same transparency as integers are: the size of the value accessed by a variable may vary arbitrarily, and there is no concept of pointers or sharing in the semantics (yet the system will prevent any unnecessary copies from being made internally). The language is in principal statically typed, allowing type errors to be caught before one is in the middle of a computation, but because of the interpreted nature of the language, typing proceeds in a somewhat special order. There are no type declarations (except for function parameters), and types of variables are deduced from the values assigned to them, but function bodies are not type–checked when they are defined, in order to allow mutually recursive functions; instead each function is type checked whenever a command is issued that will directly or indirectly invoke the function. Because of the typing regime the language can be liberal with respect to operator and function overloading: there may be many instances defined of an operator symbol or function name, as long as they can be distinguished by number and/or type of their arguments (the reader may have noticed that this is used already for the built–in operators and functions, such as '*' and *dim*).

As has been pointed out before, the library of built–in functions are not written in the LiE programming language. However this language is well suited for writing programs to solve problems which are less basic than those addressed by the built–in functions, but which use them as computational building blocks. A large collection of such programs (often quite short and simple) to solve commonly encountered problems is provided with the package, and is explained in detail in the manual. There is also a growing collection of programs that have been developed by several users of LiE; information can be obtained from either the authors or the distributors of LiE (see below). Because the source code is provided with LiE, it is even possible to write particularly time–critical programs in 'C' and add them to the library of LiE.

Although the fact that the data types of LiE are few and simple is certainly a limitation, it can also sometimes be an advantage. The data structures provided by the language are the same as those required by the basic algorithms, and they are stored very efficiently. Indeed, the vectors and matrices of the LiE language are basically just 'C' arrays and matrices of integers, with a little bit of wrapping to suit the programming language and memory management requirements. Therefore sometimes LiE can successfully tackle problems whose sheer size precludes application of many other packages, provided of course that the data required is representable in LiE; one particular example where we experienced this was Gaussian elimination of a large (non–sparse) system of linear equations over a prime field.

3.3 Other components

Apart from the library and the programming language there are a few more components that we mention for completeness. Of course there are useful utilities at a basic level which are needed to support the more visible features of the program, but which one seldomly has to deal with directly: for instance the memory management system, the type checking system and the system that binds the function calls written by the user to actual routines present in the library, resolving any overloading that may be present.

More interestingly the interactive environment has a help system which provides two kinds of information: the `help` command explains the proper use of functions and programming language, while the `learn` command gives theoretical background information on the subject of Lie groups and their representations, which can be helpful in understanding the

way to interpret the inputs and outputs of the functions mathematically. On certain systems an input line editor is provided to facilitate command re–entry.

4 Availability

The package is available on many platforms, and could moreover be easily ported to any computer system with a C–compiler. Platforms on which the program is directly available include a wide variety of workstations under UNIX–like operating systems, VAX/VMS, IBM PC compatibles, Apple Macintosh, Amiga and Atari ST. Since the distribution includes the source code, users may make local adaptations to the program should this be desired. The price of LiE depends on the desired platform and license, but as a very rough indication the price for a copy of LiE comes somewhere between $200 and $300 at the time of writing; this includes the manual and shipping costs. Distribution of LiE is performed by the CAN Expertise Centre, which can also be contacted for more information and up–to–date details about availability and price, at

CAN Expertise Centre Kruislaan 413,
1098 SJ Amsterdam,
The Netherlands
Phone: +31 20 5926050,
Fax: +31 20 5924199
Email: lie@can.nl

5 References

[Hum] Humphreys, J. E. Introduction to Lie algebras and representation theory Springer, New York 1972

[Lee] Leeuwen, M. A. A. van, Cohen, A. M., Lisser, B. LiE, A package for Lie group computations Computer Algebra Nederland, Amsterdam 1992