SUPERCALC

A REDUCE package for commutator calculations

Werner M. Seiler [1)]

Institut für Theoretische Physik, Universität Karlsruhe [2)]

D-7500 Karlsruhe 1, West Germany

BITNET: BE04@DKAUNI2

Abstract

A REDUCE package for commutator calculations in supersymmetric theories (including ordered products) and for infinite sums is presented and an application to the computation of anomalies in string theory is given.

[1)] Supported by Studienstiftung des deutschen Volkes

[2)] New address: Institut für Algorithmen und Kognitive Systeme, Universität Karlsruhe

Program Summary

- *Computers for which the program is designed and operable:* Any computer with an implementation of REDUCE 3.3. Tested on IBM 3090 with TSO, Siemens 7881 with TSS and SUN 3/60 with UNIX.
- *Operating system:* See above.
- *Programming language used:* REDUCE 3.3 (RLISP)
- *High speed storage required:* Depends on the complexity of the calculations.
- *No. of lines in combined program and test deck:* ca. 4000
- *Keywords:* Bracket computation in (super) Lie and Poisson algebras, ordered products and the theorem of Wick, infinite sums, computer algebra
- *Nature of physical problem:* Symbolic computation of commutators, handling of ordered products, evaluation of infinite sums
- *Method of solution:* Using the algebraic properties of bracket structures, the commutator (or Poisson bracket) of complicated operators is expressed by the commutators of fundamental operators. For ordered products, the theorem of Wick is applied. Infinite sums are simplified by an heuristic approach.
- *Typical running times:* Depends on the complexity of the expressions; usually between a few seconds and a few minutes.
- *Unusual features:* Applicable to a wide range of problems; due to close integration into the REDUCE system fast.

1 Introduction

The package *SUPERCALC*, written in REDUCE 3.3 [1], arose out of a work in superstring theory. To perform a BRST quantization of the different supersymmetric string models, requires the evaluation of many commutators between operators defined as infinite sums over ordered products. Doing these calculations by hand is a boring, tedious and especially error prone task. Therfore, it is a natural thought to leave them to a computer algebra system like REDUCE.

Up to now, applications of computer algebra to high energy physics are mainly concerned with the computations of Feynman graphs. In model calculations and connected problems, however, only few authors used computer algebra systems: Castellani reports about using REDUCE in supergravity [2], other authors tried it in superspace formalism [3,4]. This situation differs considerably from the situation in general relativity, where the use of computer algebra has become fairly common [5].

The reason is easy to unterstand. In general relativity most calculations are within the framework of exterior calculus. This formalism, which is well suited for computers, has been implemented in many systems (e.g. the EXCALC package [6] for REDUCE). Hence, it is possible to perform computer aided computation without being forced to write large programs.

In model calculations, however, often elements from many different branches of mathematics come together. Many of them are not implemented in computer algebra systems. Only for group theory, a larger number of programs exist, but they are mostly written in languages like PASCAL or FORTRAN.

In this paper, we present the package *SUPERCALC* for commutator calculations in supersymmetric theories, for handling ordered products, and for simplification of infinite sums. As an application, the critical dimensions of string and superstring are determined by computing the anomalies (or Schwinger terms) of their constraint algebras. Cecchini and Tarlani [7] recently presented a COMMON LISP program for commutator calculations in Lie and Poisson algebras. But in our case, the integration into a general purpose computer algebra system like REDUCE is important for further processing of the results, especially of the infinite sums.

The paper is organized as follows: Section 2 shows the physical problem sketching the basic facts about string and BRST theory. The next two sections describe the usage of *SUPERCALC* and present some sample calculations. Details of the package are given in the following three sections. First, we consider commutator calculations and the theorem of Wick. Then, section 6 deals with the simplification of infinite sums, followed by the discussion of several other aspects of the package. In the last section, some conclusions are drawn. Several comments on problems with field operators and formal derivatives are given in appendix A. Appendix B contains a complete list of the *SUPERCALC* procedures being accessible within the algebraic mode. Throughout this paper, REDUCE commands are written in upper case, *SUPERCALC* procedures in lower case.

2 String Theory

In the last years, string theory [8] has attracted much interest as a possible candidate for a unified theory. A striking feature of this theory is the existence of a so-called critical dimension D. It turns out, that quantization can be performed consistently only in a certain dimension of space-time. The critical dimensions of the different models have been determined with many different techniques; in this paper, we use a mode expansion and the BRST formalism.

Strings are one-dimensional objects. During their time evolution, they sweep out a world sheet, a two-dimensional manifold embedded in space-time. As a natural extension of point dynamics, we take the area of the world sheet as action. The basic fields of the bosonic theory are therefore the embedding functions X^μ ($\mu = 0, \ldots, D-1$) of the world sheet. Our fundamental operators will be their Fourier modes α_m^μ, satisfying the canonical commutation relation

$$[\alpha_m^\mu, \alpha_n^\nu] = m\eta^{\mu\nu}\delta_{m+n,0} \,. \tag{2.1}$$

Due to the reparametrization invariance of the action, the Lagrangian is degenerate and as a constraint the energy-momentum tensor must vanish. Its Fourier modes

$$L_m = \frac{1}{2}\sum_{-\infty}^{\infty} \alpha^{\mu}_{m-n}\alpha_{n\mu} \tag{2.2}$$

generate the well-known Virasoro algebra

$$[L_m, L_n] = (m-n)L_{m+n}\,. \tag{2.3}$$

A straightforward quantization leads to an indefinite Fock space. To take care of the constraints, we use the BRST formalism [9]. It introduces for each constraint L_m a "ghost" c_m with opposite grading and its conjugate momentum $\bar{c}_m$ obeying the canonical algebra

$$[c_m, c_n]_+ = [\bar{c}_m, \bar{c}_n]_+ = 0\,, \tag{2.4a}$$

$$[c_m, \bar{c}_n]_+ = \delta_{m+n,0}\,. \tag{2.4b}$$

The BRST operator given by

$$\mathcal{Q} = \sum L_m c_{-m} - \frac{1}{2}\sum (m-n) c_{-m}c_{-n}\bar{c}_{m+n} \tag{2.5}$$

is nilpotent,

$$Q^2 = \frac{1}{2}\,[\mathcal{Q},\mathcal{Q}]_+ = 0\,, \tag{2.6}$$

and defines the physical states with its cohomology classes. We further introduce the so-called extended constraints

$$\hat{L}_m := [\mathcal{Q}, \bar{c}_m] = L_m + \sum (m+n)\bar{c}_{m-n}c_n\,. \tag{2.7}$$

As usual, we must choose an ordering. The simplest choice is normal ordering defined by the pairings:

$$\underbrace{\alpha^{\mu}_{m}\alpha^{\nu}_{n}} = m\theta(m)\eta^{\mu\nu}\delta_{m+n,0}\,, \tag{2.8a}$$

$$\underbrace{c_m\bar{c}_n} = \theta(m)\delta_{m+n,0}\,. \tag{2.8b}$$

But now the commutators (2.3) and (2.6) acquire anomalies. As consistency condition, we demand either the vanishing of the anomaly in (2.6) or, equivalently, that the extended constraints (2.7) satisfy the Virasoro algebra (2.3) without an anomaly[3]. These conditions determine the space-time dimension D uniquely.

To commute two ordered products, we need the theorem of Wick [10]. It expands a product into a sum of ordered products. We are interested in the case, that two ordered products are multiplied:

$$\begin{aligned} :A_1 \cdots A_n::B_1 \cdots B_m: \; &= \; :A_1 \cdots A_n B_1 \cdots B_m: + \\ &+ \sum_{\text{one pairing}} :A_1 \cdots \not{A_i} \cdots A_n B_1 \cdots \not{B_j} \cdots B_m: + \\ &+ \sum_{\text{two pairings}} :A_1 \cdots A_i \cdots A_k \cdots A_n B_1 \cdots B_l \cdots B_j \cdots B_m: + \; \ldots \end{aligned} \tag{2.9}$$

3 Using *SUPERCALC*

Actually, i It is not necessary to make use of a computer algebra system in the bosonic case outlined in the last section. However, the amount of calculation grows significantly when we consider superstrings, especially in the case of models with extended supersymmetry [11,12]. Even for these, it is still possible to perform the computations by hand. But they become very tedious and the main problem is to get the right result; computer algebra ensures here the correctness.

In *SUPERCALC*, operators are introduced with the procedures `bosonic` and `fermionic`, resp.:

```
bosonic A,B,C;
```

Both procedures automatically declare their arguments as `NONCOM`muting `OPERATOR`s. So, the user does not need to give these declarations. The commutation relations of the fundamental operators are defined with `LET` rules for the operator `commutator`[4]:

```
LET commutator(A(),B()) = C();
```

[3] This equivalence, which facilitates the calculations significantly, holds only for constraint algebras of rank 1 [9]. The membrane, for example, yields a rank 2 algebra.

[4] REDUCE does not know noncommuting *variables*. One must always use operators, hence we write `A()` instead of `A`

(see also figure 1 for the declarations for the bosonic string). As the symmetry properties of a commutator are known to the system, it is not necessary to enter both, $[A, B]$ and $[B, A]$. If the switch `zerocomm` is set (which is the default in *SUPERCALC*), only the nonvanishing commutators must be given. Any commutator without a defining `LET` rule is automatically eliminated.

Commutator calculations are performed by the procedure `bracket`. It uses only the three basic algebraic properties of a graded commutator:

$$[A, B] = (-)^{\epsilon_A \epsilon_B} [B, A] \ , \tag{3.1a}$$

$$[A + B, C] = [A, C] + [B, C] \ , \tag{3.1b}$$

$$[AB, C] = A\,[B, C] + (-)^{\epsilon_B \epsilon_C} [A, C]\, B \, , \tag{3.1c}$$

where ϵ_A denotes the grading of the operator A. Hence, `bracket` can be used for any bracket structure obeying these rules, in particular for Poisson or Jacoby brackets.

In our calculations, all occuring operator products are normal ordered. In this case, `bracket` cannot apply the rule (3.1c). Instead, it follows the definition of a commutator:

$$[\ :A:, :B:\ ] = \ :A::B: \ - \ :B::A: \ . \tag{3.2}$$

The products on the right hand side are evaluated using the theorem of Wick (2.9).

Evaluating the theorem of Wick is a very tedious task. But *SUPERCALC* provides two procedures for this: `wick` expands a product in a sum of ordered products and `ordprod` does the same for a product of ordered products. *SUPERCALC* supports two orderings. For normal ordering, denoted by `normord`, the user must introduce the pairings of the noncommuting operators. Similiar to the commutation relations, this is done with `LET` rules, but now for the operator `pairing`:

```
LET pairing(A(),B()) = C();
```

As pairings possess a priori no symmetry, both, $\underline{AB}$ and $\underline{BA}$ must be given. But *SUPERCALC* knows, that the pairing of two (anti)commuting operators vanishes. Weyl ordering — total (anti)symmetrisation — is denoted by `weylord`. Here the pairing is just half of the commutator. For example, `wick(A()*B())` yields `:A()*B():+C()`, whereas `ordprod(normord(A()*B()),A())` results in `:A()^2*B():+:A()*C():`.

SUPERCALC introduces the operator `ssum` for sums. It takes four arguments: `ssum(s,n,l,u)` means $\sum_{n=l}^{u} s$. If a bound is infinity, the keyword `aleph` is used. Whereas simple sums, that means with a summand which is constant, linear or quadratic in the summation index, are automatically evaluated, the user must call the procedure `evalsum` to simplify more involved sums. For example, `evalsum(a*ssum(b*sin(n),n,-aleph,aleph))` yields zero.

For a better legible output, *SUPERCALC* provides the declaration `doindex`. After `doindex A,B,C;`, all arguments of the operators `A`, `B`, and `C` are written as subscripts. In many cases, this yields a considerable improvement of the output. The declaration can be cancelled by a call of the procedure `offindex`. Additionally, *SUPERCALC* writes ordered products and commutators following the usual conventions (colons for normal ordering and square brackets for commutators).

The so far presented procedures are the most important ones for calculations. Besides these, *SUPERCALC* contains many others. Appendix B gives a complete list of all procedures, which can be called within the algebraic mode of REDUCE.

4 Examples

As first example, we consider the bosonic string theory, as it was outlined in section 2. The first step is to introduce the fundamental modes α_m^μ and $c_n, \bar{c}_n$. The corresponding declarations are shown in figure 1. We distinguish c_n and $\bar{c}_n$ by an additional argument of the operator `c`. The space-time index μ on the matter modes is suppressed, because all expressions are completely contracted.

```
bosonic alpha;                 % definition of the modes
fermionic c;
FACTOR alpha,c;
doindex alpha,c;               % index notation for the modes
FOR ALL m,n LET
  commutator(c(1,m),c(0,n))=delta(m+n,0),
  commutator(alpha(m),alpha(n))=delta(m+n,0)*m,
  pairing(c(1,m),c(0,n))=delta(m+n,0)*theta(m),
  pairing(c(0,m),c(1,n))=delta(m+n,0)*theta(m),
  pairing(alpha(m),alpha(n))=delta(m+n,0)*theta(m)*m;
```

Figure 1. Commutation relations and pairings for the bosonic string as defined in (2.1), (2.4) and (2.8). Only the nonvanishing commutators are given. The `FACTOR` declaration improves the readibility of the output.

Figure 2 shows a complete session calculating the matter sector with anomaly. Due to the normal ordering, the Virasoro algebra has now acquired a central term:

$$[L_m, L_n] = (m-n)L_{m+n} + \frac{D}{12}(m^3 - m)\delta_{m+n,0} \,. \tag{4.1}$$

Because of the suppression of the space-time indices, the dimension D does not appear in the final result. The calculation splits in two steps: The first one is to perform the commutator calculation and is done completely automatic. The second and more complicated one consists of the simplification of the result. Here the user must give some hints.

```
setgreater(m,0);
Time: 3 ms
pp:=bracket(normord(alpha(m-j)*alpha(j)),
            normord(alpha(n-k)*alpha(k)));
PP := DELTA           *DELTA     *THETA(-J+M)*THETA(J)*J*(-J+M)
          -J-K+M+N,0       J+K,0
      +DELTA           *DELTA     *THETA(-K+N)*THETA(K)*K*(K-N)
           -J-K+M+N,0       J+K,0
      +DELTA           *THETA(-J+M)*:ALPHA *ALPHA :*(-J+M)
           -J-K+M+N,0                    J      K
      +DELTA           *THETA(-K+N)*:ALPHA *ALPHA :*(K-N)
           -J-K+M+N,0                    J      K
      +DELTA         *DELTA        *THETA(-J+M)*THETA(J)*J*(-J+M)
           -J+K+M,0       J-K+N,0
      +DELTA         *DELTA        *THETA(-K+N)*THETA(K)*K*(K-N)
           -J+K+M,0       J-K+N,0
      +DELTA         *THETA(-J+M)*:ALPHA     *ALPHA :*(-J+M)
           -J+K+M,0                    -K+N       J
      -(DELTA         *THETA(K)*K)*:ALPHA     *ALPHA :
            -J+K+M,0                    -K+N       J
      +DELTA        *THETA(-K+N)*:ALPHA     *ALPHA :*(K-N)
           J-K+N,0                    -J+M       K
      +DELTA        *THETA(J)*J*:ALPHA     *ALPHA :
           J-K+N,0                   -J+M       K
      +DELTA      *THETA(J)*J*:ALPHA     *ALPHA     :
           J+K,0                   -J+M       -K+N
      -(DELTA      *THETA(K)*K)* :ALPHA     *ALPHA     :
            J+K,0                     -J+M       -K+N
Time: 1378 ms
for all x such that not freeof(x,n) let
  delta(m+n,0)*x=delta(m+n,0)*sub(n=-m,x);
Time: 15 ms
qq:=evalsum ssum(pp,j,-aleph,aleph);
QQ := -(2*DELTA      *THETA(-K)*THETA(K+M)*K)*(K+M)
               M+N,0
      +2*DELTA      *THETA(-K-M)*THETA(K)*K*(K+M)
              M+N,0
      -(THETA(-K)*K)*:ALPHA     *ALPHA    :
                           -K+N      K+M
      -(THETA(-K)*K)*:ALPHA    *ALPHA     :
                           K+M       -K+N
      +2*THETA(-K+N)*:ALPHA       *ALPHA :*(K-N)
                           -K+M+N      K
      +2*THETA(K-N)*:ALPHA       *ALPHA :*(K-N)
```

```
                          -K+M+N      K
     -(THETA(K)*K)*:ALPHA     *ALPHA   :
                          -K+N      K+M
     -(THETA(K)*K)*:ALPHA    *ALPHA    :
                          K+M      -K+N
Time: 3068 ms
rr:=evalsum ssum(qq,k,-aleph,aleph);
                       2
RR := (DELTA     *M*(M -1)
            M+N,0
       +6*SSUM(:ALPHA        *ALPHA   :*I:1,I:1,-ALEPH,ALEPH)
                     M+N-I:1       I:1
       -3*SSUM(:ALPHA      *ALPHA      :*I:1,I:1,-ALEPH,ALEPH)
                     M+I:1      N-I:1
       -3*SSUM(:ALPHA      *ALPHA      :*I:1,I:1,-ALEPH,ALEPH)
                     N-I:1      M+I:1
       -(6*N)*SSUM(:ALPHA        *ALPHA   :,I:1,-ALEPH,ALEPH))/3
                         M+N-I:1       I:1
Time: 2934 ms
for all k let
  ssum(normord(alpha(m+k)*alpha(n-k))*k,k,-aleph,aleph)=
    ssum(normord(alpha(m+n-k)*alpha(k))*(k-m),k,-aleph,aleph);
Time: 32 ms
rr/4;
                 2
(DELTA     *M*(M -1)
      M+N,0
 +6*SSUM(:ALPHA        *ALPHA   :,I:1,-ALEPH,ALEPH)*(M-N))/12
               M+N-I:1      I:1
Time: 275 ms
```

Figure 2. A sample session computing the Virasoro algebra (4.1). The output is slightly edited (some blanks removed and lines reordered). The times refer to a Siemens 7881 mainframe with TSS; an IBM 3090 has approximately double speed. A SUN 3/60 with UNIX needs between seven and ten times longer, but the whole calculation is still done in less than a minute.

The first line declares m to be greater than zero. We must distinguish different cases here, because the collection of some intermediate expressions depends on that; but the final results show no dependency. The next step computes the commutator of two Virasoro operators (2.2). Before we sum over the dummy indices, we give *SUPERCALC* a hint, how to handle products with $\delta_{m+n,0}$. This help is necessary to achieve the simplest form of the result. (This rule entered *before* `bracket` is called slows down the calculation by 50%!) Some index shifts must be performed, before the mode expansion of L_{m+n} can be recognized. They are again implemented with `LET` rules. Indices of the form `i:<number>` are generated by *SUPERCALC*.

As second example, we compute the fermionic contribution to the algebra of the super Virasoro operators for the $N = 1$ superstring. They are defined by

$$L_m = \frac{1}{2} \sum_{n \in \mathbb{Z}} :\alpha^\mu_{m-n} \alpha_{n\mu}: \; + \frac{1}{2} \sum_{n \in \mathbb{Z}+\frac{1}{2}} (n - \frac{m}{2}) : d^\mu_{m-n} d_{n\mu} : \tag{4.2}$$

and again satisfy the algebra (2.2). The modes d^μ_m bear half integer indices and also obey canonical commutation relations:

$$[d^\mu_m, d^\nu_n]_+ = \eta^{\mu\nu}\delta_{m+n,0}\,. \tag{4.3}$$

Their ordering is defined by $\underbracket{d^\mu_m d^\nu_n} = \theta(m)\eta^{\mu\nu}\delta_{m+n,0}$. These properties are introduced to the system analogously to figure 1. Again we suppress the world sheet indices.

Because of the similiarity to the first example, figure 3 shows only the input lines and the final result. The first command declares the indices as half integer. The ghost contributions are calculated exactly the same way. All put together yields the well-known values of $D = 26$ and $D = 10$ for the bosonic and the $N = 1$ superstring, resp. Similarly, one can compute the algebras of the extended constraints and so the critical dimensions for the models with extended supersymmetry. In each case, the number of modes and hence the size of the algebras doubles. The results are $D = 2$ for $N = 2$ and $D = -2$ for $N = 4$ in accordance with [11].

```
halfind j,k;
Time: 3 ms
for all x such that not freeof(x,n) let
  delta(m+n,0)*x=delta(m+n,0)*sub(n=-m,x);
Time: 16 ms
setgreater(m,0);
Time: 4 ms
pp:=bracket((j-m/2)*normord(d(m-j)*d(j)),
           (k-n/2)*normord(d(n-k)*d(k)));
Time: 8435 ms
qq:=evalsum ssum(pp,j,-aleph,aleph);
Time: 8292 ms
rr:=evalsum ssum(qq,k,-aleph,aleph)/4;
                        2
RR := (2*DELTA     *M*(M -1)
              M+N,0
                                   2
       +24*SSUM(:D       *D   :*H:2 ,H:2,-ALEPH,ALEPH)
                  M+N-H:2  H:2
       +12*SSUM(:D       *D   :*H:2,H:2,-ALEPH,ALEPH)*(-M-3*N)
                  M+N-H:2  H:2
                                   2
       +12*SSUM(:D     *D     :*H:2 ,H:2,-ALEPH,ALEPH)
                  M+H:2  N-H:2
       +6*SSUM(:D     *D     :*H:2,H:2,-ALEPH,ALEPH)*(M-N)
                 M+H:2  N-H:2
                                   2
       -12*SSUM(:D     *D     :*H:2 ,H:2,-ALEPH,ALEPH)
                  N-H:2  M+H:2
       +6*SSUM(:D     *D     :*H:2,H:2,-ALEPH,ALEPH)*(-M+N)
                 N-H:2  M+H:2
       +6*SSUM(:D       *D   :,H:2,-ALEPH,ALEPH)*N*(M+2*N)
                 M+N-H:2  H:2
       -(3*M*N)*SSUM(:D     *D     :,H:2,-ALEPH,ALEPH)
                       M+H:2  N-H:2
```

```
        +3*M*N*SSUM(:D      *D      :,H:2, - ALEPH,ALEPH))/48
                     N-H:2  M+H:2
Time: 9415 ms
```

Figure 3. Fermionic contribution to the algebra of the super Virasoro operators. The indices are now half integers. There exist no easy way to simplify the final result with LET rules.

In the example of figure 2, it would be easily possible to reexpress the result as a Virasoro operator with the help of a LET rule. In figure 3 we encounter a different situation. The necessary index shifts are much more tricky here[5)]. The result can be simplified to $\frac{m-n}{2}\sum_{j\in\mathbb{Z}+\frac{1}{2}}(j-\frac{m+n}{2}) : d_{m+n-j}d_j :$. But to find the shifts requires to have an idea of the form of the final expression. It is for instance crucial to know that no quadratic term in j but a factor $(m-n)$ should occur. Furthermore, one must split terms and perform different shifts on the parts. The whole operation is too complex to be worthwhile programming, if an algorithmic approach exists at all.

Besides, the most important result is always the anomaly. The structure constants of the algebras can be more or less guessed. In the example of figure 3, they follow already from the bosonic part which was calculated in figure 2. For the anomalies, one can only derive their form (which powers of m are occurring) with the help of the Jacoby identity [12] but not the exact expressions. `evalsum` was able to simplify all occurring scalar sums; all anomalies came out in the simplest possible form.

5 Commutator Calculations and the theorem of Wick in *SUPERCALC*

In principle, `bracket` uses a simple recursive implementation of the rules (3.1): The arguments are decomposed, until both are fundamental operators. Then the `commutator` of these is returned to be evaluated by the LET rules given by the user. But this leads to a fairly inefficient algorithm, because many calculations are executed several times, as it can be seen from the simple example $[A+B+C,O]$, where O is some complicated expression. An application of rule (3.1a) yields $[A,O]+[B,O]+[C,O]$, which

5) Further problems arise from the fact, that the REDUCE pattern matcher does not work correctly with noncommuting quantities. This can be seen already from figure 2, where it does not lead to an error, because bosonic operators can be interchanged in an ordered product.

means that O must be decomposed three times. Such redundancy cannot be avoided completely; the same happens in hand calculations.

`bracket` tries two strategies to improve the efficiency. First, it computes the complexity (defined as the number of needed recursive calls) of each argument and starts with the more complicated one. But the decisive point is that for the most common cases – both arguments are either sums, products or powers – special procedures are invoked. These implement a formula for the corresponding special case, which can be computed iteratively.

`bracket` can handle any expression built arithmetically with the exception of the division by an operator. Hence, the inverse of an operator has to be introduced separately. This range can be enlarged in two ways: An operator can be declared `first` (get the flag `first` in its property list). This means, that physical operators occur only in its first argument and that the operator is linear in this argument. So the commutator can be interchanged with the operator. As default, the REDUCE operators `DF` (differentiation) and `INT` (integration) and the *SUPERCALC* summation operator `ssum` bear this flag.

For special operations (e.g. a new ordered product), a user can provide own procedures which are automatically invoked by `bracket`. For this purpose, the name of the procedure must be `PUT` under the `special` indicator. If the property list of an operator contains an entry with this indicator, its value is interpreted as the name of a function with two arguments that will perform the computation of the commutator. The function can be written in either mode, symbolic or algebraic.

The theorem of Wick is implemented in two procedures: `wick` expands a product into a sum of ordered expressions, `ordprod` performs the same for the product of two ordered products as shown in (2.9). As mentioned in section 3, `ordprod` is used by `bracket`, if ordered terms appear. Both procedures apply a depth-first algorithm to generate all pairings. This algorithm has a high complexity: For a small number of factors, the number of terms in the expansion grows exponentially, asymptotically the complexity is even n^n. For an ordered product with five factors we get 25 terms, with six already 75 and with seven 231 and with eight 763 summands[6], resp.

[6] Seven or eight factors can easily occur. The BRST quantization of the membrane requires for example to commute two ordered expressions with five factors. This yields 1545 terms!

For further computations ordered terms must be transformed into a canonical representation to allow for the identification of identical terms. This task is performed by `simpord`. It takes scalar or c-number factors out of the ordering and rearranges the operators in a standard order (induced by `ORDP`). For this purpose, `simpord` applies a simple sorting routine which takes care of the signs generated by commuting fermionic operators. `wick` and `ordprod` call `simpord` automatically, so their results are always normalized.

6 Simplification of Sums

SUPERCALC makes no use of summation theory. It uses a completely heuristic approach. The choice of the implemented rules followed from the need of the intended calculations: They suffice to compute all anomalies arising in the BRST quantization of string models.

The simplification takes place in two steps. For elementary sums it is integrated into the REDUCE simplifier. These sums – at present: summand constant, linear or quadratic in the summation index – are evaluated by a kind of table look-up. At the same level, the linearity of `ssum` is implemented. This cannot be done with a `LINEAR` declaration, because we must often work with noncommuting summands and this case is not treated by REDUCE.

More involved sums are taken care of by `evalsum`. This procedure implements a rule based algorithm using six rules to simplify formal sums. The first three concern single sums. These are first checked for an even or odd symmetry. Then, sums with a Kronecker δ are evaluated at once, if `SOLVE` can detect equal arguments of δ within the summation range. Eventually, the bounds of sums containing step functions of the form $\theta(const \pm n)$ are adjusted. Here the usual convention of string theory $\theta(0) = \frac{1}{2}$ is adopted.

The three other rules work on expressions linear in `ssum`. They try to reduce the number of sums. First,`evalsum` looks for sums with ranges of equal size. They are collected into one sum in the hope that REDUCE finds a simplification of the new summand. Then, sums with equal summands are collected, if either they cancel each other partially or their ranges are adjacent. Often these rules can be applied only after an index shift. `evalsum` tries to deduce such shifts from the bounds of the sums and performs them automatically then.

To make the evaluation process more efficient, a special data structure is used within `evalsum`. Any expression linear in `ssum` is transformed into a list which allows easy access to all relevant parameters. Nevertheless, simplifying sums consumes a lot of time, especially for multiple sums where each level must be considered separately.

The main reason for this inefficency is the control strategy for applying the rules. It turned out to be surprisingly tricky. The problem is that we must always test all rules for both forms $\sum a_i + \sum b_i$ and $\sum(a_i + b_i)$, if a collection is possible. Hence, it often happens that a sum is tested several times for an application of the same rule. After each change, all rules must be checked again. But this is the drawback of any rule based system.

7 Some remarks

Sections 5 and 6 described the main algorithms of *SUPERCALC*. But beside these a lot of small r problems have to be tackled. They add up to one third of the total code of the package! This effect is probably typical for a symbolic mode program and the main disadvantage of using RLISP. REDUCE is a completely open system at this level, the programmer has access to and can redefine everything. But on the other hand, the symbolic mode does not offer much programming comfort.

As a first point, we need the procedures `setless` and `setgreater` to declare a variable to be greater or less than a given number. For instance in the example of figure 2, the step functions in some intermediate sums can be handled only, if it is known, whether some bounds are greater or less than zero. Now we must of course extend the relations `less` and `greater`, so that they make use of the declarations. These new procedures can also handle infinity.

In the Neveu-Schwarz sector of the superstring the indices of the fermionic modes are half integers $\pm\frac{1}{2}, \pm\frac{3}{2}, \ldots$ Hence, we need a procedure (`halfind`) to declare the type of indices and another one (`halfp`) to decide, whether a given expression yields an integer or half integer result (or neither). For a term of the form $\frac{i+j}{2}$, it must then be known whether the denominator is even or odd. This kind of information is stored in the property list of the variables. Further procedures generate automatically identifiers of a given type with a unique name. They are used by `evalsum`, as one can see from figures 2 and 3.

A well-known problem in computer algebra consists in the legibility of the output. Especially casual users often can hardly recognize their results. On a standard terminal with an ASCII character set, not many possibilities exist to improve this situation. *SUPERCALC* tries nevertheless to stick as close as feasible to human conventions. The most important topic here is index notation for which an old algorithm of Hulshof and van Hulzen [13] for REDUCE 3.2 was adapted.

As usual, normal ordering is denoted by colons and remaining commutators are enclosed in square brackets. This can be easily achieve by putting a corresponding output function in the property list of the operators. The sums, especially multiple sums, remains as main problem. As they are written in an one-dimensional format in the output, they are difficult to survey. The only satisfactory solution consists probably in a connection to a REDUCE-TEX-interface (e.g. TRI [14]) or something similar.

Mostly for "historical" reasons (earlier versions of *SUPERCALC* relied heavily on `LET` rules), a simple mechanism for switching groups of `LET` rules on and off is implemented. `LET` rules offer fairly flexible possibilities, but they decrease the speed considerably. Especially `bracket` shows a great sensitivity to this problem as mentioned in the discussion of the first example. In *SUPERCALC*, the user has to write two procedures: One contains the definitions of the rules, the other one the `CLEAR` commands. After a call of `letswitch` with the name of the group as argument, the rules are invoked and cleared with `ON <groupname>` and `OFF <groupname>`, resp. A more "professional" solution to this problem with more possibilities was recently included into the REDUCE e-mail library.

As main data structure, REDUCE uses the so-called standard quotients. It implements a recursive, sparse representation of polynomials. Although this structure possesses many advantageous features, it does not suit very well our purposes. If, for instance in a commutator calculation, an expression of the form `A*B*C` is appearing, all factors should be treated on the same footing. This is much easier realized with a LISP prefix form (TIMES A B C) than with a complicated recursive dotted pair structure. Therefore, all procedures in *SUPERCALC* work with prefix forms. This slows down the simplification slightly, because transformations between the different formats must be executed.

8 Conclusions

The examples show that a completly automatic computation of the anomalies is not possible. To get a sensible and simple form of the output, hints in form of `LET` rules must be given. Therefore, interactive use of the system is necessary, which requires reasonable execution times. In REDUCE, we can achieve this only by working in the symbolic (RLISP) mode and by a close integration into the REDUCE system. Neither can be done without a detailed knowledge of the internal structure of REDUCE (about which no documentation beside the source code exists). At least the most important flags and properties used by simplifier, parser, and output routines have to be known.

In the supersymmetric string models all commutators can be decomposed in a sequence of computations like the ones presented in section 3. It is a characteristic feature of this approach, that only terms bilinear in the modes are occurring. This represents the decisive advantage of using the extended constraints. A direct calculation of $\mathcal{Q}^2$ makes much more difficulties: At first sight, one might think that four ghost terms of the form

$$\sum_{m,n,l} (m-n)(m+n-l) : c_{-m} c_{-n} c_{-l} \bar{c}_{m+n+l} :$$

appear in the commutator. But these expressions vanish due to symmetry. The proof requires nontrivial manipulations of the sums: We must perform cyclic permutations of the summation indices and add the generated terms. Such tricks are far beyond the capability of a simple procedure like `evalsum`.

To commute two bilinear terms as shown in the examples takes not too much time, even in hand calculations. So the main reason to use a computer algebra lies in the correctness of the results. A large number of similiar computations represents a permanent source of errors. Especially the frequent switches from bosonic to fermionic modes very easily lead to wrong signs.

In a recent paper, Gorman *et al.* [15] derived fairly general formulae for the commutators of bilinear currents like (2.2) and their anomalies. But an application of these results proved to be fairly tedious and an implementation would require a highly nontrivial pattern matching. Besides, such a program would be very specialiced, whereas *SUPERCALC* can be used for any calculation in which brackets, ordered products or sums are occuring. Hence, a computer aided, brute force approach seems to be superior here.

A system like *SUPERCALC* might allow to handle membrane theory with the same formalism. Up to now, it has been possible to calculate by hand the anomaly of a truncated version of the BRST algebra only [16]. But for the membrane, no mode expansion is possible, the fields themselves must be used. This leads to some problems (see appendix A). Another possible application consists in calculating the Lorentz algebra of either string or membrane in the light-cone gauge [7]. Anyway, *SUPERCALC* should be able to calculate nearly any commutator or Poisson bracket. In most cases, the problems will start after the call of `bracket`, because the results will need further evaluation. We had to handle infinite sums for instance.

Appendix A. Field operators

If we want to perform similar calculations for the membrane, we cannot use a mode expansion but must work with the fields themselves. `bracket` has no problems with this, but afterwards we need integration (especially of Green's functions and of the δ distributions) instead of summation. Of course, also derivatives of the fields occur. The `DF` and `INT` procedures of REDUCE are designed for concrete calculations. In formal computations the results are in no normal or canonical representation. E.g., the expression `DF(f(x-y),x)+DF(f(x-y),y)` is not simplified to zero.

This example shows at once the problem and the solution: We must introduce a new operator `diff` which uses the chain rule and distinguishes between two different notations: `diff(f(x**2,y),1,1)` means that the function `f` is differentiated once with respect to its first argument and then evaluated at the point (x^2, y). With `diff(f(x**2,y),x,1)` the function `f` is differentiated once with respect to `x` yielding `2*x*diff(f(x**2,y),1,1)`.

This notation leads easily to errors, because it is not well readable. One can no longer omit the number of differentiations, because this would lead to ambiguities. But we get a canonical representation and `diff(f(x-y),x,1)+diff(f(x-y),y,1)` simplifies to zero. A package implementing this syntax is in preparation. It will also include the necessary procedures for integration with Green's functions and the δ distribution. Then all necessary tools for calculations with fields are gathered and handling the membrane might become possible.

Appendix B. User accessible procedures

This appendix summarizes all procedures of *SUPERCALC* which can be called in the algebraic mode. They are listed with their syntax and a short description. All other procedures are only accessible in the symbolic mode, but they are mainly auxiliary functions and therefore of minor interest.

SUPERCALC introduces further three new flags or indicators: `first`, `grade`, and `special`. The global variables of the system all carry the escape character `!:` in their names. Hence, a user should avoid such name.

`bosonic a,b,...`	The arguments `a,b,...` are declared as bosonic, noncommuting operators.
`bracket(x,y)`	Computes the graded commutator of `x` and `y`.
`doindex a,b,...`	The arguments of the operators `a,b, ...` will be printed as subscripts.
`evalsum x`	Simplifies the sums in `x`.
`expord x`	The ordered product `x` is expanded in unordered products.
`fermionic a,b,...`	The arguments are declared as fermionic, noncommuting operators.
`first a,b,...`	The arguments are the names of operators being declared `first` and `NONCOM`.
`grade x`	Calculates the grade of `x`.
`greater(x,y)`	Checks, whether `x` is strictly greater than `y`.
`halfind a,b,...`	The arguments are declared as half integer indices.
`halfp x`	Checks, whether the expression `x` yields an integer or half integer result.
`intind a,b,...`	The arguments are declared as integer indices.
`jacoby(x,y,z)`	The Jacoby identity of `x,y,z` is computed.
`less(x,y)`	Checks, whether `x` is strictly less than `y`.
`letswitch a,b,...`	Declares `a,b,...` as switches for defining and clearing `LET` rules.
`maxi x`	The maximum of the list `x` is determined using `greater`.
`mini x`	The minimum of the list `x` is determined using `less`.

`normalize x`	Summation indices get normalized names, to allow for the recognition of identical expressions.
`oddp x`	Predicate. True, if `x` yields an odd result.
`offindex a,b,...`	The arguments of the operators `a,b`, ... are no longer considered as indices.
`ordprod(x,y)`	Wick expansion of the product of two ordered products.
`seteven a,b,c,...`	The arguments are declared as even numbers.
`setgreater(x,y)`	Declares `x` to be strictly greater than `y`. One argument must be a number.
`setless(x,y)`	Declares `x` to be strictly less than `y`. One argument must be a number.
`setodd a,b,c,...`	The arguments are declared as odd numbers.
`simpord x`	Scalar factors are taken out of ordered terms and the operators are written in a standard order (defined by `ORDP`).
`wick x`	Wick expansion of a product.

9 Acknowledgments

It's a pleasure to thank Göktürk Üçoluk for many valuable hints about REDUCE, and Marcus Scholl for many discussions about BRST and string theory.

10 References

[1] A.C. Hearn: REDUCE 3.3 – User Manual, *RAND Publication CP78, The RAND Corporation, Santa Monica 1987*

[2] L. Castellani, *Int. J. Mod. Phys. A3(1988)1435*

[3] R. Grimm, H. Kühnelt, *Comp. Phys. Comm. 20(1980)77*

[4] R.P. dos Santos, *J. Symb. Comp. 7(1989)523*

[5] I. Cohen, I. Frick, J.E. Åman, *In:* Proc. 10th Int. Conf. General Relativity and Gravitation, *B. Bertotti et al. (eds.), Dordrecht 1984, p. 139*

[6] E. Schrüfer, F.W. Hehl, J.D. McCrea, *Gen. Rel. Grav. 19(1987)197*

[7] R. Cecchini, M. Tarlani, *Comp. Phys. Comm. 52(1989)283*

[8] M.B. Green, J.H. Schwarz, E. Witten: Superstring Theory, vol. I&II, *Cambridge University Press, Cambridge (UK) 1987*

[9] M. Henneaux, *Phys. Rep. 126(1985)1*

[10] N.N. Bogoljubov, D.V. Shirkov: Introduction to the Theory of Quantized Fields, *Interscience Publisher, New York 1959*

[11] S.D. Mathur, S. Mukhi, *Phys. Rev. D36(1987)465*

[12] W.M. Seiler: Die Bestimmung der kritischen Dimensionen von String und Superstring mit REDUCE, *master thesis (in german), Karlsruhe 1989*

[13] B.J.A. Hulshof, J.A. van Hulzen: Some REDUCE Facilities for Pretty Printing Subscripts and Formal Derivatives, *Technische Hogeschool Twente Memorandum Nr. INF-82-11, Enschede 1982*

[14] W. Antweiler, A. Strotmann, V. Winkelmann: Typesetting REDUCE Output with TEX – A REDUCE-TEX-Interface –, *internal paper, Rechenzentrum der Universität zu Köln, 1989*

[15] N. Gorman, W. McGlinn, L. O'Raifeartaigh, D. Williams, *Int. J. Mod. Phys. A4(1989)1235*

[16] M. Scholl: Über die kritischen Dimensionen höherdimensionaler relativistischer Objekte, *thesis (in german), Karlsruhe 1988*