PHYSOP
A Package for Operator Calculus in Quantum Theory

User's Manual
Version 1.5
January 1992

Mathias Warns
Physikalisches Institut der Universität Bonn
Endenicher Allee 11–13
D–5300 BONN 1
Germany

Tel: (++49) 228 733724
Fax: (++49) 228 737869
e–mail: UNP008@DBNRHRZ1.bitnet

1 Introduction

The package PHYSOP has been designed to meet the requirements of theoretical physicists looking for a computer algebra tool to perform complicated calculations in quantum theory with expressions containing operators. These operations consist mainly in the calculation of commutators between operator expressions and in the evaluations of operator matrix elements in some abstract space. Since the capabilities of the current REDUCE release to deal with complex expressions containing noncommutative operators are rather restricted, the first step was to enhance these possibilities in order to achieve a better usability of REDUCE for these kind of calculations. This has led to the development of a first package called NONCOM2 which is described in section 2. For more complicated expressions involving both scalar quantities and operators the need for an additional data type has emerged in

order to make a clear separation between the various objects present in the calculation. The implementation of this new REDUCE data type is realized by the PHYSOP (for PHYSical OPerator) package described in section 3.

2 The NONCOM2 Package

The package NONCOM2 redefines some standard REDUCE routines in order to modify the way noncommutative operators are handled by the system. In standard REDUCE declaring an operator to be noncommutative using the `NONCOM` statement puts a global flag on the operator. This flag is checked when the system has to decide whether or not two operators commute during the manipulation of an expression.

The NONCOM2 package redefines the `NONCOM` statement in a way more suitable for calculations in physics. Operators have now to be declared noncommutative pairwise, i.e. coding:

```
NONCOM A,B;
```

declares the operators `A` and `B` to be noncommutative but allows them to commute with any other (noncommutative or not) operator present in the expression. In a similar way if one wants e.g. `A(X)` and `A(Y)` not to commute, one has now to code:

```
 NONCOM A,A;
```

Each operator gets a new property list containing the operators with which it does not commute. A final example should make the use of the redefined `NONCOM` statement clear:

```
NONCOM A,B,C;
```

declares `A` to be noncommutative with `B` and `C`, `B` to be noncommutative with `A` and `C` and `C` to be noncommutative with `A` and `B`. Note that after these declaration e.g. `A(X)` and `A(Y)` are still commuting kernels.

Finally to keep the compatibility with standard REDUCE declaring a single identifier using the `NONCOM` statement has the same effect as in standard REDUCE i.e., the identifier is flagged with the `NONCOM` tag.

From the user's point of view there are no other new commands implemented

by the package. Commutation relations have to be declared in the standard way as described in the manual i.e. using `LET` statements. The package itself consists of several redefined standard REDUCE routines to handle the new definition of noncommutativity in multiplications and pattern matching processes.

CAVEAT: Due to its nature, the package is highly version dependent. The current version has been designed for the 3.3 and 3.4 releases of REDUCE and may not work with previous versions. Some different (but still correct) results may occur by using this package in conjunction with LET statements since part of the pattern matching routines have been redesigned. The package has been designed to bridge a deficiency of the current REDUCE version concerning the notion of noncommutativity and it is the author's hope that it will be made obsolete by a future release of REDUCE.

3 The PHYSOP package

The package PHYSOP implements a new REDUCE data type to perform calculations with physical operators. The noncommutativity of operators is implemented using the NONCOM2 package so this file should be loaded prior to the use of PHYSOP[1]. In the following the new commands implemented by the package are described. Beside these additional commands, the full set of standard REDUCE instructions remains available for performing any other calculation.

3.1 Type declaration commands

The new REDUCE data type PHYSOP implemented by the package allows the definition of a new kind of operators (i.e. kernels carrying an arbitrary number of arguments). Throughout this manual, the name "operator" will refer, unless explicitly stated otherwise, to this new data type. This data type is in turn divided into 5 subtypes. For each of this subtype, a declaration command has been defined:

`SCALOP A;` declares `A` to be a scalar operator. This operator may carry an arbitrary number of arguments i.e. after the declaration: `SCALOP A;`

[1] To build a fast loading version of PHYSOP the NONCOM2 source code should be read in prior to the PHYSOP code

all kernels of the form e.g. `A(J), A(1,N), A(N,L,M)` are recognized by the system as being scalar operators.

`VECOP V;` declares `V` to be a vector operator. As for scalar operators, the vector operators may carry an arbitrary number of arguments. For example `V(3)` can be used to represent the vector operator $\vec{V}_3$. Note that the dimension of space in which this operator lives is arbitrary. One can however address a specific component of the vector operator by using a special index declared as `PHYSINDEX` (see below). This index must then be the first in the argument list of the vector operator.

`TENSOP C(3);` declares `C` to be a tensor operator of rank 3. Tensor operators of any fixed integer rank larger than 1 can be declared. Again this operator may carry an arbitrary number of arguments and the space dimension is not fixed. The tensor components can be addressed by using special `PHYSINDEX` indices (see below) which have to be placed in front of all other arguments in the argument list.

`STATE U;` declares `U` to be a state, i.e. an object on which operators have a certain action. The state U can also carry an arbitrary number of arguments.

`PHYSINDEX X;` declares `X` to be a special index which will be used to address components of vector and tensor operators.

It is very important to understand precisely the way how the type declaration commands work in order to avoid type mismatch errors when using the PHYSOP package. The following examples should illustrate the way the program interprets type declarations. Assume that the declarations listed above have been typed in by the user, then:

- `A,A(1,N),A(N,M,K)` are SCALAR operators.
- `V,V(3),V(N,M)` are VECTOR operators.
- `C, C(5),C(Y,Z)` are TENSOR operators of rank 3.
- `U,U(P),U(N,L,M)` are STATES.

BUT: `V(X),V(X,3),V(X,N,M)` are all scalar operators since the special index `X` addresses a specific component of the vector operator (which is a scalar operator). Accordingly, `C(X,X,X)` is also a scalar operator because the diagonal component C_{xxx} of the tensor operator `C` is meant here (C has rank 3 so 3 special indices must be used for the components).

In view of these examples, every time the following text refers to scalar operators, it should be understood that this means not only operators defined by the SCALOP statement but also components of vector and tensor operators. Depending on the situation, in some case when dealing only with the components of vector or tensor operators it may be preferable to use an operator declared with SCALOP rather than addressing the components using several special indices (throughout the manual, indices declared with the PHYSINDEX command are referred to as special indices).

Another important feature of the system is that for each operator declared using the statements described above, the system generates 2 additional operators of the same type: the adjoint and the inverse operator. These operators are accessible to the user for subsequent calculations without any new declaration. The syntax is as following:

If A has been declared to be an operator (scalar, vector or tensor) the adjoint operator is denoted A!+ and the inverse operator is denoted A!-1 (an inverse adjoint operator A!+!-1 is also generated). The exclamation marks do not appear when these operators are printed out by REDUCE (except when the switch NAT is set to off) but have to be typed in when these operators are used in an input expression. An adjoint (but no inverse) state is also generated for every state defined by the user. One may consider these generated operators as "placeholders" which means that these operators are considered by default as being completely independent of the original operator. Especially if some value is assigned to the original operator, this value is not automatically assigned to the generated operators. The user must code additional assignement statements in order to get the corresponding values.

Exceptions from these rules are (i) that inverse operators are always ordered at the same place as the original operators and (ii) that the expressions A!-1*A and A*A!-1 are replaced[2] by the unit operator UNIT . This operator is defined as a scalar operator during the initialization of the PHYSOP package. It should be used to indicate the type of an operator expression whenever no other PHYSOP occur in it. For example, the following sequence:

```
SCALOP A;
A:= 5;
```

[2]This may not always occur in intermediate steps of a calculation due to efficiency reasons.

leads to a type mismatch error and should be replaced by:

```
SCALOP A;
A:=5*UNIT;
```

The operator `UNIT` is a reserved variable of the system and should not be used for other purposes.

All other kernels (including standard REDUCE operators) occurring in expressions are treated as ordinary scalar variables without any PHYSOP type (referred to as scalars in the following). Assignement statements are checked to ensure correct operator type assignement on both sides leading to an error if a type mismatch occurs. However an assignement statement of the form `A:= 0` or `LET A = 0` is always valid regardless of the type of `A`.

Finally a command `CLEARPHYSOP` has been defined to remove the PHYSOP type from an identifier in order to use it for subsequent calculations (e.g. as an ordinary REDUCE operator). However it should be remembered that no substitution rule is cleared by this function. It is therefore left to the user's responsibility to clear previously all substitution rules involving the identifier from which the PHYSOP type is removed.

Users should be very careful when defining procedures or statements of the type `FOR ALL ... LET ...` that the PHYSOP type of all identifiers occurring in such expressions is unambigously fixed. The type analysing procedure is rather restrictive and will print out a "PHYSOP type conflict" error message if such ambiguities occur.

3.2 Ordering of operators in an expression

The ordering of kernels in an expression is performed according to the following rules:
1. Scalars are always ordered ahead of PHYSOP operators in an expression. The REDUCE statement `KORDER` can be used to control the ordering of scalars but has no effect on the ordering of operators.

2. The default ordering of operators follows the order in which they have been declared (and not the alphabetical one). This ordering scheme can be changed using the command `OPORDER`. Its syntax is similar to the `KORDER` statement, i.e. coding: `OPORDER A,V,F;` means that all occurrences of the

operator A are ordered ahead of those of V etc. It is also possible to include operators carrying indices (both normal and special ones) in the argument list of OPORDER. However including objects not defined as operators (i.e. scalars or indices) in the argument list of the OPORDER command leads to an error.

3. Adjoint operators are placed by the declaration commands just after the original operators on the OPORDER list. Changing the place of an operator on this list means not that the adjoint operator is moved accordingly. This adjoint operator can be moved freely by including it in the argument list of the OPORDER command.

3.3 Arithmetic operations on operators

The following arithmetic operations are possible with operator expressions:

1. Multiplication or division of an operator by a scalar.

2. Addition and subtraction of operators of the same type.

3. Multiplication of operators is only defined between two scalar operators.

4. The scalar product of two VECTOR operators is implemented with a new function DOT . The system expands the product of two vector operators into an ordinary product of the components of these operators by inserting a special index generated by the program. To give an example, if one codes:

```
VECOP V,W;
V DOT W;
```

the system will transform the product into:

```
V(IDX1) * W(IDX1)
```

where IDX1 is a PHYSINDEX generated by the system (called a DUMMY INDEX in the following) to express the summation over the components. The identifiers IDXn (n is a nonzero integer) are reserved variables for this purpose and should not be used for other applications. The arithmetic operator DOT can be used both in infix and prefix form with two arguments.

5. Operators (but not states) can only be raised to an integer power. The

system expands this power expression into a product of the corresponding number of terms inserting dummy indices if necessary. The following examples explain the transformations occurring on power expressions (system output is indicated with an `-->`):

```
SCALOP A; A**2;
- --> A*A
VECOP V; V**4;
- --> V(IDX1)*V(IDX1)*V(IDX2)*V(IDX2)
TENSOP C(2); C**2;
- --> C(IDX3,IDX4)*C(IDX3,IDX4)
```

Note in particular the way how the system interprets powers of tensor operators which is different from the notation used in matrix algebra.

6. Quotients of operators are only defined between scalar operator expressions. The system transforms the quotient of 2 scalar operators into the product of the first operator times the inverse of the second one. Example[3]:

```
SCALOP A,B;   A / B;
       -1
 --> (B  )*A
```

7. Combining the last 2 rules explains the way how the system handles negative powers of operators:

```
SCALOP B;
B**(-3);
       -1    -1    -1
 --> (B  )*(B  )*(B  )
```

The method of inserting dummy indices and expanding powers of operators has been chosen to facilitate the handling of complicated operator expressions and particularly their application on states (see section 3.4.3). However it may be useful to get rid of these dummy indices in order to enhance the readability of the system's final output. For this purpose the switch `CONTRACT` has to be turned on (`CONTRACT` is normally set to `OFF`). The system in this case contracts over dummy indices reinserting the `DOT` operator and reassembling the expanded powers. However due to the prede-

[3]This shows how inverse operators are printed out when the switch `NAT` is on

fined operator ordering the system may not remove all the dummy indices introduced previously.

3.4 Special functions

3.4.1 Commutation relations

If 2 PHYSOPs have been declared noncommutative using the (redefined) NONCOM statement, it is possible to introduce in the environment elementary (anti-) commutation relations between them. For this purpose, 2 scalar operators COMM and ANTICOMM are available. These operators are used in conjunction with LET statements. Example:

```
SCALOP A,B,C,D;
LET COMM(A,B)=C;
FOR ALL N,M LET ANTICOMM(A(N),B(M))=D;
VECOP U,V,W; PHYSINDEX X,Y,Z;
FOR ALL X,Y LET COMM(V(X),W(Y))=U(Z);
```

Note that if special indices are used as dummy variables in FOR ALL ... LET constructs then these indices should have been declared previously using the PHYSINDEX command.

Every time the system encounters a product term involving 2 noncommutative operators which have to be reordered on account of the given operator ordering, the list of available (anti-) commutators is checked in the following way: First the system looks for a commutation relation which matches the product term. If it fails then the defined anticommutation relations are checked. If there is no successful match the product term A*B is replaced by:

```
A*B;
 --> COMM(A,B) + B*A
```

so that the user may introduce the commutation relation later on.

The user may want to force the system to look for anticommutators only; for this purpose a switch ANTICOM is defined which has to be turned on (ANTICOM is normally set to OFF). In this case, the above example is replaced by:

```
ON ANTICOM;
A*B;
 -->  ANTICOMM(A,B) - B*A
```

Once the operator ordering has been fixed (in the example above `B` has to be ordered ahead of `A`), there is no way to prevent the system from introducing (anti-)commutators every time it encounters a product whose terms are not in the right order. On the other hand, simply by changing the `OPORDER` statement and reevaluating the expression one can change the operator ordering without the need to introduce new commutation relations. Consider the following example:

```
SCALOP A,B,C;   NONCOM A,B;   OPORDER B,A;
LET COMM(A,B)=C;
A*B;
- -->  B*A + C;
OPORDER A,B;
 B*A;
- --> A*B - C;
```

The functions `COMM` and `ANTICOMM` should only be used to define elementary (anti-) commutation relations between single operators. For the calculation of (anti-) commutators between complex operator expressions, the functions `COMMUTE` and `ANTICOMMUTE` have been defined. Example (is included as example 1 in the test file):

```
VECOP P,A,K;
PHYSINDEX X,Y;
FOR ALL X,Y LET COMM(P(X),A(Y))=K(X)*A(Y);
COMMUTE(P**2,P DOT A);
```

3.4.2 Adjoint expressions

As has been already mentioned, for each operator and state defined using the declaration commands quoted in section 3.1, the system generates automatically the corresponding adjoint operator. For the calculation of the adjoint representation of a complicated operator expression, a function `ADJ`

has been defined. Example[4]:

```
SCALOP A,B;
ADJ(A*B);
       +    +
 --> (B )*(A )
```

3.4.3 Application of operators on states

For this purpose, a function `OPAPPLY` has been defined. It has 2 arguments and is used in the following combinations:

(i) `LET OPAPPLY(`*operator, state*`) =` *state*; This is to define a elementary action of an operator on a state in analogy to the way elementary commutation relations are introduced to the system. Example:

```
SCALOP A; STATE U;
FOR ALL N,P LET OPAPPLY((A(N),U(P))= EXP(I*N*P)*U(P);
```

(ii) `LET OPAPPLY(`*state, state*`) =` *scalar exp.*; This form is to define scalar products between states and normalization conditions. Example:

```
STATE U;
FOR ALL N,M LET OPAPPLY(U(N),U(M)) = IF N=M THEN 1 ELSE 0;
```

(iii) *state* `:= OPAPPLY(`*operator expression, state*); In this way, the action of an operator expression on a given state is calculated using elementary relations defined as explained in **(i)**. The result may be assigned to a different state vector.

(iv) `OPAPPLY(`*state*`, OPAPPLY(`*operator expression, state*`))`; This is the way how to calculate matrix elements of operator expressions. The system proceeds in the following way: first the rightmost operator is applied on the right state, which means that the system tries to find an elementary relation which match the application of the operator on the state. If it fails the system tries to apply the leftmost operator of the expression on the left state using the adjoint representations. If this fails also, the system prints out a warning message and stops the evaluation. Otherwise the next operator oc-

[4]This shows how adjoint operators are printed out when the switch `NAT` is on

curing in the expression is taken and so on until the complete expression is applied. Then the system looks for a relation expressing the scalar product of the two resulting states and prints out the final result. An example of such a calculation is given in the test file.

The infix version of the `OPAPPLY` function is the vertical bar | . It is right associative and placed in the precedence list just above the minus (−) operator. Some of the REDUCE implementation may not work with this character, the prefix form should then be used instead[5].

4 Known problems in the current release of PHYSOP

(i) Some spurious negative powers of operators may appear in the result of a calculation using the PHYSOP package. This is a purely "cosmetic" effect which is due to an additional factorization of the expression in the output printing routines of REDUCE. Setting off the REDUCE switch `ALLFAC` (`ALLFAC` is normally on) should make these terms disappear and print out the correct result (see example 1 in the test file).

(ii) The current release of the PHYSOP package is not optimized w.r.t. computation speed. Users should be aware that the evaluation of complicated expressions involving a lot of commutation relations requires a significant amount of CPU time and memory. Therefore the use of PHYSOP on small machines is rather limited. A minimal hardware configuration should include at least 4 MB of memory and a reasonably fast CPU (type Intel 80386 or equiv.).

(iii) Slightly different ordering of operators (especially with multiple occurrences of the same operator with different indices) may appear in some calculations due to the internal ordering of atoms in the underlying LISP system (see last example in the test file). This cannot be entirely avoided by the package but does not affect the correctness of the results.

[5]The source code can also be modified to choose another special character for the function

5 Compilation of the packages

To build a fast loading module of the NONCOM2 package, enter the following commands after starting the REDUCE system:

```
faslout "noncom2";
 in "noncom2.red";
faslend;
```

To build a fast loading module of the PHYSOP package, enter the following commands after starting the REDUCE system:

```
faslout "physop";
 in "noncom2.red";
 in "physop.red";
faslend;
```

Input and output file specifications may change according to the underlying operating system.
On PSL–based systems, a spurious message:

```
*** unknown function PHYSOP!*SQ called from compiled code
```

may appear during the compilation of the PHYSOP package. This warning has no effect on the functionality of the package.

6 Final remarks

The package PHYSOP has been presented by the author at the IV inter. Conference on Computer Algebra in Physical Research, Dubna (USSR) 1990 (see M. Warns, *Software Extensions of REDUCE for Operator Calculus in Quantum Theory*, Proc. of the IV inter. Conf. on Computer Algebra in Physical Research, Dubna 1990, to appear). It has been developed with the aim in mind to perform calculations of the type exemplified in the test file included in the distribution of this package. However it should also be useful in some other domains like e.g. the calculations of complicated Feynman diagrams in QCD which could not be performed using the HEPHYS package. The author is therefore grateful for any suggestion to improve or extend the

usability of the package. Users should not hesitate to contact the author for additional help and explanations on how to use this package. Some bugs may also appear which have not been discovered during the tests performed prior to the release of this version. Please send in this case to the author a short input and output listing displaying the encountered problem.

Acknowledgements

The main ideas for the implementation of a new data type in the REDUCE environnement have been taken from the VECTOR package developed by Dr. David Harper (D. Harper, Comp. Phys. Comm. **54** (1989) 295). Useful discussions with Dr. Eberhard Schrüfer and Prof. John Fitch are also gratefully acknowledged.

A List of error and warning messages

In the following the error (E) and warning (W) messages specific to the PHYSOP package are listed.

`cannot declare` *x* `as` *data type* (W): An attempt has been made to declare an object *x* which cannot be used as a PHYSOP operator of the required type. The declaration command is ignored.

x `already defined as` *data type* (W): The object *x* has already been declared using a REDUCE type declaration command and can therefore not be used as a PHYSOP operator. The declaration command is ignored.

x `already declared as` *data type* (W): The object `x` has already been declared with a PHYSOP declaration command. The declaration command is ignored.

x `is not a PHYSOP` (E): An invalid argument has been included in an `OPORDER` command. Check the arguments.

`invalid argument(s) to` *function* (E): A function implemented by the PHYSOP package has been called with an invalid argument. Check type of arguments.

`Type conflict in` *operation* (E): A PHYSOP type conflict has occured during an arithmetic operation. Check the arguments.

`invalid call of` *function* `with args:` *arguments* (E): A function of the PHYSOP package has been declared with invalid argument(s). Check the argument list.

`type mismatch in` *expression* (E): A type mismatch has been detected in an expression. Check the corresponding expression.

`type mismatch in` *assignement* (E): A type mismatch has been detected in an assignment or in a `LET` statement. Check the listed statement.

`PHYSOP type conflict in` *expr* (E): A ambiguity has been detected during the type analysis of the expression. Check the expression.

`operators in exponent cannot be handled` (E): An operator has occurred in the exponent of an expression.

`cannot raise a state to a power` (E): states cannot be exponentiated by the system.

`invalid quotient` (E): An invalid denominator has occurred in a quotient. Check the expression.

`physops of different types cannot be commuted` (E): An invalid operator has occurred in a call of the `COMMUTE/ANTICOMMUTE` function.

`commutators only implemented between scalar operators` (E): An invalid operator has occurred in the call of the `COMMUTE/ANTICOMMUTE` function.

`evaluation incomplete due to missing elementary relations` (W): The system has not found all the elementary commutators or application relations necessary to calculate or reorder the input expression. The result may however be used for further calculations.

B List of available commands

inputphysop.idx