Title

COSY INFINITY Reference Manual

Author

Berz, M

Publication Date

1990-07-01

LBL-28881
UC-405

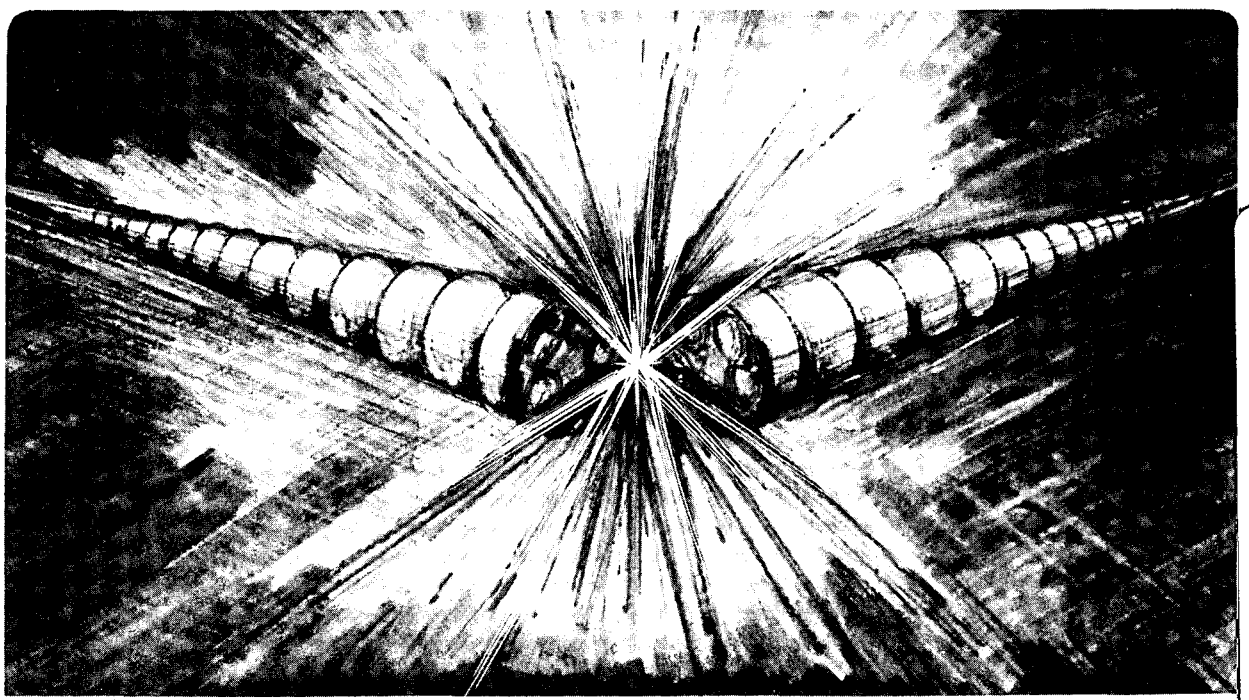
Lawrence Berkeley Laboratory
UNIVERSITY OF CALIFORNIA

Accelerator & Fusion
Research Division

COSY INFINITY Reference Manual

M. Berz

July 1990



Prepared for the U.S. Department of Energy under Contract Number DE-AC03-76SF00098.

DISCLAIMER

LBL-28881
ESG Note-103

COSY INFINITY REFERENCE MANUAL*

Martin Berz

Accelerator and Fusion Research Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, CA 94720

July 1990

*This work was supported by the Director, Office of Energy Research, Office of Basic Energy Sciences, Materials Sciences Division, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

COSY INFINITY
Reference Manual

M. Berz

Accelerator and Fusion Research Division
Lawrence Berkeley Laboratory
1 Cyclotron Road
Berkeley, CA 94720

Abstract

This is a reference manual for the arbitrary order particle optics and beam dynamics code COSY INFINITY. It is current as of June 28, 1990. COSY INFINITY is a code to study and design particle optical systems, including beamlines, spectrometers, and particle accelerators. At its core it is using differential algebraic (DA) methods, which allow a very systematic and simple calculation of high order effects. At the same time, it allows the computation of dependences on system parameters, which is often interesting in its own right and can also be used for fitting.

COSY INFINITY has a full structured object oriented language environment. This provides a simple interface for the casual user. At the same time, it offers the demanding user a very flexible and powerful tool for the study and design of systems, and more generally, the utilization of DA methods.

The power and generality of the environment is perhaps best demonstrated by the fact that the physics routines of COSY INFINITY are written in its own input language and are very compact. The approach also considerably facilitates the implementation of new features because they are incorporated with the same commands that are used for design and study.

Contents

1 How to Obtain and Use the Code

The code to COSY INFINITY can be requested from us at the address on the cover of this manual. It can be obtained by email, or on various tapes and floppies. In the latter case, the user is requested to furnish the media.

Users are requested not to make the code available to others, but ask them to obtain it from us. We want to maintain a list of users to be able to send out regular updates; in particular, the structure of the COSY language is hoped to foster self-growth, and all users should benefit from this.

Though we do our best to keep the code bug free and hope that it is so now, we do not mind being convinced of the contrary and ask users to report any errors. Users are also encouraged to make suggestions for upgrades, or send us their tools written in the COSY language.

If the user thinks the code has been useful for a project, we would like to see this acknowledged by referencing some of the papers related to the code, for example [1, 2]. Finally, we do neither guarantee correctness nor usefulness of this code, and we are not liable for any damage, material or emotional, that results from its use.

2 How to Install the Code

The code for COSY INFINITY consists of the following files:

- FOXY.FOP
- DAFOX.FOP
- FOXFIT.FOP
- COSY.FOX

If obtained by electronic mail, each of these files will be split into several pieces. The pieces can be identified by the filename followed by the part number in the subject; for example, DAFOX.FOP003 identifies the third part of the file DAFOX. The files then have to be reassembled before compilation.

FOXY.FOP is the compiler and executer of the COSY language and is written in FORTRAN 77. DAFOX.FOP contains the routines to perform elementary operations, in particular the differential algebraic routines, written in FORTRAN 77. FOXFIT.FOP contains the package of nonlinear optimizers, written in FORTRAN 77. These three files have to be compiled and linked. COSY.FOX contains all the physics of COSY INFINITY, and is written in its own input language. It is compiled by FOXY.

COSY INFINITY is written in standard FORTRAN 77, and unusual constructs are avoided as far as possible. However, certain things in FORTRAN 77 are still system dependent; in particular, this concerns the file handling. COSY INFINITY contains the proper code for various machines, including VAX/VMS, SUN/UNIX, IBM, and CRAY (which is not fully complete at this time).

This is achieved by selectively commenting and un-commenting certain lines. To go from VAX to SUN, for example, all lines that have the identified *VAX somewhere in columns 72 through 80 have to be commented, and all lines that have the comment *SUN in columns 1 through 4 have to be uncommented. There is a utility FORTRAN program called VERSION that performs all these changes automatically. Should the user still experience problems, a short message to us would be appreciated in order to facilitate life for future users on the same system.

There may also be some memory limitations. Being based on FORTRAN which does not allow dynamic memory allocation COSY INFINITY has its own memory management that is based on a large COMMON block. On machines supporting virtual memory, the size of this block should not represent any problem. On other machines, it may be necessary to scale the length down. This can be achieved by changing the parameter LMEM at all occurrences in FOXY.FOP and DAFOX.FOP to the same value. Values of around 100000 should be enough for many applications, which brings total system memory down to a little over 1 Megabyte.

In the case of limited memory resources, it may be necessary to scale down the lengths of certain variables in COSY.FOX to lower levels. In particular, this holds for the variables MAP and SCR which are defined at the very beginning of COSY.FOX. Possible values for the length are values down to about 500 for work through around fifth order. For higher orders, larger values are needed.

3 Introduction and Philosophical Questions

From the very beginning, the design and analysis of particle optical systems has been quite intimately connected with the computer world. There are numerous more or less widespread codes for the simulation of particle optical systems. Generally, these codes fall into two categories. One category includes ray tracing codes which use numerical integrators to determine the trajectories of individual rays through external and possibly internal electromagnetic fields. The core of such a code is quite robust and easy to set up; for some applications, however, the mere values of final positions of rays is not enough information. Furthermore, this type of code usually is quite slow and does not allow extensive optimization.

The other category of codes are the mapping codes, which compute Taylor expansions to describe the action of the system on phase space. These codes usually are much faster than integration codes, and the expansion coefficients often provide more insight into the system. On the other hand, the orders of the map, which are a measure of the accuracy of the approach, have been limited to third order [3, 4, 5] and fifth order [6, 7]. Furthermore, traditional mapping codes had only very limited libraries for quite standardized external fields and lacked the flexibility of the numerical integration techniques.

Recently we could show that it is indeed possible to have the best of both worlds: using the new differential algebraic techniques, any given numerical integration code can be modified such that it allows the computation of Taylor maps for arbitrarily complicated fields and to arbitrary order [8, 9, 10, 11]. An offspring of this approach is the computation of maps for large accelerators where often the system can be described by inexpensive, low order integrators [12, 13].

The speed of this approach is determined by the numerical integration process. Recently it has been possible to use DA techniques to overcome this problem: DA can be used to automatically emulate numerical integrators of very high order in the time step, yet at the computational expense of only little more than a first order integrator [8, 9]. This technique is very versatile, works for a very large class of fields, and the speeds obtained are those of classical mapping codes. Hence these techniques seem very promising for

new generation mapping codes.

In order to make efficient use of DA operations in a computer environment, it has to be possible to invoke the DA operations from a language. In the conventional languages used for numerical applications it is not possible to introduce new data types and operations on them. Only recently have object oriented languages been developed which routinely have such features. One such language which is slowly gaining ground is C++; Forth is another example which has been around for a longer time.

There are strong reasons to stay within the limits of a FORTRAN environment, however. Firstly, virtually all software in this field is written in this language, and the desire to interface to such software almost requires the use of FORTRAN. Furthermore, there are extensive libraries of support software which are only slowly becoming available in other languages, including routines for nonlinear optimization and various graphics packages. Finally, the necessity for portability is another strong argument for FORTRAN; virtually every machine that is used for numerical applications, starting from personal computers, continuing through the workstation and mainframe world through the supercomputers, has a FORTRAN compiler.

So it seemed natural to stay within this world, and this lead to the development of the DA precompiler [14, 15]. This precompiler allows the use of a DA data type within otherwise standard FORTRAN by transforming arithmetic operations containing DA variables into a sequence of calls to subroutines. This technique has been extensively used [9, 10, 16, 17, 18, 19]. It was particularly helpful that one could use old FORTRAN tracking codes and just replace the appropriate real variables by DA variables to very quickly obtain high order maps.

On the other end of the problems using an accelerator code is the command language of the code and the description of the beamlines. Various approaches have been used in the past, starting from coding numbers as in the old versions of TRANSPORT [3] over more easily readable command structures like in TRIO [4], GIOS [5, 20], COSY 5.0 [6, 17] and MARYLIE [21] to probably the most complete, standardized commands of MAD [22].

4 General Properties of the COSY Language Environment

Probably the best way to allow the user flexibility is to provide him with an environment that in itself is a full programming language. Then he can express any problem that can be expressed on a computer within this environment. Furthermore, if the code itself is written in the same language, it is rather easy even for the users to add new features to the code by themselves, allowing the code to grow naturally.

The question is which language to select. For ease of use, it should have a very simple syntax. For the user demanding special-purpose features, it should be powerful. It should allow direct and complex interfacing to FORTRAN routines, and it should allow the use of DA as a type. Finally, it should be widely portable.

Unfortunately, there is no language readily available that fulfills all these requirements. So we decided to develop our own language environment.

The problem of simplicity yet power has been quite elegantly solved by the PASCAL concept. In addition, this concept allows compilation in one pass and no time consuming linking is required. This facilitates the connection of the user input, which will turn out to be just the last procedure of the system, with the optics program itself.

To be machine independent, the output of the compilation is not machine code but rather an intermediate code that can be easily interpreted. For the same reason, it is essential to write the source code of the compiler in a very portable language. We chose FORTRAN even for this task, even though it is considerably easier to write such a compiler in a recursive language.

Details about the language syntax are described in a language description data file which is read by a program that updates the compiler. The first entry in this data file is a list of the names of all data types. The second entry is a list containing the elementary operations, information for which combinations of data types they are allowed, and the names of individual FORTRAN routines to perform the specific operations.

The third entry contains all the intrinsic functions and the types of their results. The fourth entry finally contains a list of FORTRAN procedures that can be called from the environment.

All these data are read from a program that updates the compiler; in particular, it includes all the intrinsic operations, functions and procedures into the routine that interprets the intermediate code.

For reasons of speed it is helpful to allow the splitting of the program into pieces, one containing the optics program and one the user commands. While the PASCAL philosophy does not have provisions for linking, it allows the splitting of the input at any point. For this purpose, a complete momentary image of the compilation status is dumped to a file. When compilation continues with the second portion, this image is read from the file, and compilation continues in exactly the same way as without the splitting.

5 The Syntax of the COSY Language

In this section we will discuss the syntax of the COSY language. It will become apparent that the language has the flavor of PASCAL, which is particularly easy to learn yet quite powerful and relatively easy to analyze by a compiler.

The language of COSY differs from PASCAL in its object oriented features. New data types and operations on them can easily be implemented by putting them into the language description file described above. Furthermore, all type checking is done at run time, not at compile time. This has significant advantages for the practical use of DA and will be discussed below.

Throughout this section, curly brackets like "{" and "}" denote elements that can be repeated.

Most commands of the COSY language consist of a keyword, followed by expressions and names of variables, and terminated by a semicolon. The individual entries and the semicolon are separated by blanks. The exceptions are the assignment statement, which does not have a keyword but is identified

by the assignment identifier :=, and the call to a procedure, in which case the procedure name is used instead of the keyword.

Line breaks are not significant; commands can extend over several lines, and several commands can be in one line. To facilitate readability of the code, it is possible to include comments. Everything contained within a pair of curly brackets "{" and "}" is ignored.

Each keyword and each name consist of up to 32 characters, of which the first has to be a letter and the subsequent ones can be letters, numbers or the underline sign "_" . The case of the letters is not significant.

The language consists of a tree-structured arrangement of nested program segments. There are three types of program segments. The first is the main program, which begins at the top of the input file and ends at the end. It is enclosed between the keywords

BEGIN ;

and

END ;

The other two types of program segments are procedures and functions. Their beginning and ending are denoted by the commands

PROCEDURE <name> { <name > } ;

and

ENDPROCEDURE ;

as well as

FUNCTION <name> {<name > } ;

ENDFUNCTION ;

The first name identifies the procedure and function for the purpose of calling it. The optional names define the local names of variables that are passed into the routine. Like in other languages, the name of the function can be used in arithmetic expressions, whereas the call to a procedure is a separate statement.

Inside each program segment, there are three sections. The first section contains the declaration of local variables, the second section contains the local procedures and functions, and the third section contains the executable code. A variable is declared with the following command:

VARIABLE <name> <expression> {<expression> } ;

Here the name denotes the identifier of the variable to be declared. As mentioned above, the types of variables are free at declaration time. The next expression contains the amount of memory that has to be allocated when the variable is used. The amount of memory has to be sufficient to hold the various types that the variable can assume. To simplify the determination of the required memory, there are various functions that return the required lengths for certain types.

If the variable is to be used with indices as an array, the next expressions have to specify the different dimensions. Note the elements of an array can have different types. Thus it is possible to emulate most of the record concept found in PASCAL using arrays.

Note that different from PASCAL practice, names of variables that are being passed into a function or procedure do not have to be declared.

All variables are visible inside the program segment in which they are declared as well as in all other program segments inside it. In case a variable has the same name as one that is visible from a higher level routine, its name and dimension override the name and properties of the higher level variable of the same name for the remainder of the procedure and all local procedures.

The next section of the program segment contains the declaration of local procedures and functions. Any such program segment is visible in the segment in which it was declared and in all program segments inside the segment in which it was declared, as long as the reference is physically located below the declaration of the local procedure. Recursive calls are permitted. Altogether, the local and global visibility of variables and procedures follows standard structured programming practice.

The third and final section of the program segment contains executable statements. Among the permissible executable statements is the assignment statement, which has the form

<variable or array element> := <expression> ;

The assignment statement does not require a keyword. It is characterized by the assignment identifier :=. The expression is a combination of variables and array elements visible in the routine, combined with operands, following common practice. Note that due to the object oriented features, various operands can be loaded for various data types, and default hierarchies for the operands can be given. Parentheses are allowed to override default hierarchies. The indices of array elements can themselves be expressions.

Another executable statement is the call to a procedure. This statement does not require a keyword either. It has the form

<procedure name> {<expression>} ;

The name is the identifier of the procedure to be called which has to be visible at the current position. The rest are the arguments passed into the procedure. The number of arguments has to match the number of arguments in the declaration of the procedure.

Besides the assignment statement and the procedure statement, there are statements that control the program flow. These statements consist of matching pairs denoting the beginning and ending of a control structure and sometimes of a third statement that can occur between such beginning and ending statements. Control statements can be nested as long as the beginning and ending of the lower level control structure is completely contained inside the same section of the higher level control structure.

The first such control structure begins with

IF <expression> ;

which later has to be matched by the command

ENDIF ;

If desired, there can be an arbitrary number of statements of the form

ELSEIF <expression> ;

between the matching **IF** and **ENDIF** statements.

If there is a structure involving **IF**, **ELSEIF** and **ENDIF**, the first expression in the **IF** or **ELSEIF** that is of logical type and has value true, if there is one, is determined. If there is no such expression, execution is continued after the **ENDIF**; otherwise, the section following the first such expression up to the next **ELSEIF** or **ENDIF** is executed, after which execution is continued after the **ENDIF** statement. So at most one of the sections of code separated by **IF** and the matching optional **ELSEIF** and the **ENDIF** statements is executed.

The next such control structure consists of the pair

WHILE <expression> ;

and

ENDWHILE ;

If the expression is of type logical and has the value true execution is continued after the **WHILE** statement; otherwise, it is continued after the **ENDWHILE** statement. In the former case, execution continues until the **ENDWHILE** statement is reached. After this, it continues at the matching **WHILE**, where again the expression is checked. Thus, the block is run through over and over again as long as the expression has the proper value.

Another such control structure is the familiar loop, consisting of the pair

LOOP <name> <expression> <expression> {<expression >} ;

and

ENDLOOP ;

Here the first entry is the name of a visible variable which will act as the loop variable, the first and second expressions are the first and second bounds of the loop variable. If a third expression is present, this is the step size; otherwise, the step size is set to 1. Initially the loop variable is set to the first bound.

If the step size is positive or zero and the loop variable is not greater than the second bound, or the step size is negative and loop variable is greater than the second bound, execution is continued at the next statement,

otherwise after the matching **ENDLOOP** statement. When the matching **ENDLOOP** statement is reached after execution of the statements inside the loop, the step size is added to the loop variable. Then, the value of the loop variable is compared to the second bound in the same way as above, and execution is continued after the **LOOP** or the **ENDLOOP** statement, depending on the outcome of the comparison. Note that it is allowed to alter the value of the loop variable inside the loop, which allows a premature truncation of the loop.

The final control structure in the syntax of the COSY language allows nonlinear optimization as part of the syntax of the language. This is an unusual feature not found in other languages, and it could also be expressed in other ways using procedure calls. But the great importance of nonlinear optimization in applications of the language and the clarity in the code that can be achieved with it seemed to justify such a step. The structure consists of the pair

FIT <name> {<name>} ;

and

ENDFIT <expression> <expression> <expression> ;

Here the names denote the visible variables that are being adjusted. The first expression is of real or integer type and denotes the objective quantity, the quantity that has to be minimized. The second expression is the tolerance to which the minimum is requested. The third expression gives the number of the optimizing algorithm that is being used. Currently, there is only the simplex algorithm, denoted by the value 1.

This structure is run through over and over again, where for each pass the optimization algorithm changes the values of the variables listed in the **FIT** statement and attempts to minimize the objective quantity. This continues until the algorithm does not succeed in decreasing the objective quantity anymore by more than the tolerance. After the optimization terminates, the variables contain the values corresponding to the lowest value of the objective quantity encountered by the algorithm.

Besides the commands just presented, there are commands for i/o. They appear as commands and not as procedure calls because they have variable

number of arguments. They have the form

READ <expression> <name> ;

and

WRITE <expression> {<expression>} ;

Here the first expression stands for a unit number; unit 6 is the screen. In the READ command, the name denotes the variable to be read. Note that right now only real numbers can be read. In the WRITE command, the expressions following the output unit are the output.

6 Error Messages

COSY distinguishes between four different kinds kinds of error messages which have different meanings and require different actions to correct the problem. The four types of error messages are identified by the symbols **###**, **$$$**, **!!!** and **@@@**. Their meaning is as follows.

###: This kind of message denotes errors in the syntax of the user input. Usually a short message describing the problem is given, including the command in error. If this is not enough information to remedy the problem, the file <inputfile>.lis can be consulted. It contains an element-by-element listing of the user input, including the error messages at the appropriate positions.

$$$: This error message denotes runtime errors in a syntactically correct user input. These errors include array bound violations, type violations, missing initialization of variables, variable memory exhaustion, and illegal operations like division by zero.

!!!: This kind of message denotes exhaustion of certain internal arrays in the compiler. Since the basis of COSY is FORTRAN which is not recursive and requires a fixed memory allocation, all arrays used in the compiler have to be previously declared. This entails that in certain cases of big programs etc., the upper limits of the arrays can be reached. In such a case the user is told which parameter has to be increased. The problem can be fixed by replacing

the value of the parameter by a larger value. Note that all occurrences of the parameter have to be changed; usually the parameters occur under the same name in many subroutines.

@@@: This kind of message describes a catastrophic error, and should never occur with any kind of user input, erroneous or not. It means that COSY has found an internal error in its code by using certain self checks. In the rare case that such an error message is encountered, the user is kindly asked to contact us and submit the input deck.

7 The Ion Optics Environment of COSY INFINITY

In this section we want to describe some core features of COSY's ion optics environment. This provides the backbone for practical use in particle optics. We assume that the reader has a fundamental knowledge about particle optics, and refer to the vast literature, for example [23, 24, 25, 26].

The physics part of COSY INFINITY is written in its own input language. In this context, most commands are just calls to previously defined procedures. If desired, the user can create new commands simply by defining procedures of his own.

The user input is nothing but the last procedure in COSY INFINITY. It must be included between the statements

PROCEDURE RUN ;

and

ENDPROCEDURE ;

The last command has to be followed by the call to the procedure,

RUN ;

and the command to complete the COSY input file,

END ;

The first set of commands prepares the computation of the maps. The very first command sets up the DA tools and has to be called before any DA operations, including the computation of maps, can be executed. The command has the form

OV <order> <number of variables> ;

and the parameters are the maximum order that is to occur as well as the maximum number of independent variables. If no dependences on system parameters are requested, this is just the number of phase space variables.

The following command is used to prepare the computation of maps. It sets the transfer map (which can be found in the global array MAP) to unity. It can also be used later on to erase the map.

UM ;

The following command sets the initial energy, mass and charge of the reference particle.

RP < energy in MeV > < mass in amu > < charge in units > ;

The command

PS < PX > < PA > < PY > < PB > < PD > < PG > ;

sets half widths of the beam in the x, a, y and b directions of phase space as well as energy and mass spread of the beam. The command

BETA < BX > < BY > ;

computes the momentary values of the beta functions in x and y directions, while

WAIST < LX > < LY > ;

computes the distances to the next waist in x and y directions, and

IMAGE < LX > < LY > ;

computes the distances to the next image in x and y directions. The command

PM < unit > ;

prints the transfer matrix to unit. unit = 6 corresponds to the screen.

DL < length > ;

lets a field free drift act on the map. The commands

MQ < length > < flux density at pole tip > < aperture > ;

MH < length > < flux density at pole tip > < aperture > ;

MO < length > < flux density at pole tip > < aperture > ;

MD < length > < flux density at pole tip > < aperture > ;

MZ < length > < flux density at pole tip > < aperture > ;

let a magnetic quadrupole, sextupole, octupole, decapole or duodecapole act on the map.

MM < length > < BQ >< BH >< BO >< BD >< BZ >
< aperture > ;

lets a superimposed magnetic multipole with components BQ through BZ act on the map.

MMPOLE < length > < MA > < NMA > < aperture > ;

lets a superimposed magnetic multipole act on the map. Contrary to the previous procedure, the arguments now are the array MA of NMA multipole terms.

EQ < length > < voltage at pole tip > < aperture > ;

EH < length > < voltage at pole tip > < aperture > ;

EO < length > < voltage at pole tip > < aperture > ;

ED < length > < voltage at pole tip > < aperture > ;

EZ < length > < voltage at pole tip > < aperture > ;

lets an electric quadrupole, sextupole, octupole, decapole or duodecapole act on the map.

EM < length >< EQ >< EH >< EO >< ED >< EZ >

< aperture > ;

lets a superimposed electric multipole with components EQ through EZ act on the map.

Similar to MMPOLE is the procedure

EMPOLE < length > < EA > < NEA > < aperture > ;

The next two commands let an inhomogeneous bending magnet or electrostatic deflector act on the map:

MS < radius > < angle > < aperture >
$< n_1 >< n_2 >< n_3 >< n_4 >< n_5 >$;

ES < radius > < angle > < aperture >
$< n_1 >< n_2 >< n_3 >< n_4 >< n_5 >$;

The indices n_i describe the midplane radial field dependence which is given by

$$F(r) = F_0 \cdot [1 + \sum_{i=1}^{5} n_i \cdot (\frac{x}{r})^i] \qquad (1)$$

where r is the bending radius. The command

CB ;

changes the bending direction of bending magnets and deflectors. Initially, the bending direction is clockwise. The procedure CB changes it to counterclockwise, and each additional CB switches it to the other direction.

To address individual matrix elements rather than whole maps, there is the function

ME (<phase space variable >,< element identifier >)

For example, ME(1,12) returns the momentary value of the matrix element (x,xa).

The following function computes the Poisson bracket between two DA vectors and is essential for many Lie algebraic operations:

PB (< a >,< b>)

In this section we gave a brief list of some of the important routines in COSY INFINITY. We again want to stress that it is not the philosophy of COSY INFINITY to provide commands for all conceivable questions. Rather, we want to provide the sophisticated user with a simple environment that easily allows him to construct such commands himself. The next section will contain some examples for such situations.

8 Examples

In this section we will give some practical examples how the COSY INFINITY concept can be used in practice. We begin with a simple problem, namely the computation of the transfer map of a quadrupole dublet to tenth order. Here the COSY input resembles the input of many other optics codes [6, 5].

```
PROCEDURE RUN ;
   OV 10 4 ;        {sets order to 10 and number of variables to 4}
   RP .1 4 2 ;      {energy: 10 MeV, mass: 4 amu,  charge: 2 units}
   UM ;
   DL .1 ;                                    {drift of length .1 m}
   MQ .2 .1 .05 ;  {quadrupole of length .2 m, tip field .1 T,
   DL .1 ;                                           aperture .05 m}
   MQ .2 -.1 .05 ; {defocussing quadrupole}
   DL .1 ;
   PM 11 ;          {prints map to unit 11}
   ENDPROCEDURE ;
   RUN ; END ;
```

In the next example, we compute the map depending on the energy and mass of the particle as well as on the strength of a quadrupole. Here the intrinsic COSY function DA(n) is used, which returns the DA vector that is zero except for a 1 in slot n (compare [8, 9]). This shows how simple it is to

fully treat all system parameter dependences using DA techniques as long as types are checked only at run time.

```
PROCEDURE RUN ;
   OV 5 7 ;              {sets order to 5 and number of variables to 7}
   RP .1 4*(1+DA(5)) 2*(1+DA(6)) ;     {sets energy, mass and charge;
                                 now mass and charge are DA quantities}
   UM ;
   DL .1 ;               {drift of length .1 m}
   MQ .2 .1*(1+DA(7)) .05 ;  {quadrupole; now field is a DA quantity}
   DL .1 ;
   MQ .2 -.1 .05 ;
   DL .1 ;
   PM 11 ;               {prints map, containing dependence on energy,
                              mass and quadrupole strength, to unit 11}
   ENDPROCEDURE ;
   RUN ; END ;
```

Next we will show an example with optimization. A triplet is fitted to perform stigmatic point-to-point imaging. After the imaging is completed, the triplet is placed in series 10 times as a simple beam guidance system. The total map is written to the screen, and depending on its elements, the user can start the optimization again with different parameters. For this purpose, the intrinsic COSY procedures READ and WRITE are used.

```
PROCEDURE RUN ;
  VARIABLE V1 1 ; VARIABLE V2 1 ; VARIABLE V3 1 ;
  VARIABLE I  1 ; VARIABLE L  1 ;      {defines local variables}
  PROCEDURE TRIPLET L A B C ; {defines a section of a beamline}
    DL L ; MQ .1 A .05 ; DL .05 ;
    MQ .1 B .05 ; DL .05 ; MQ .1 C .05 ; DL L ;
  ENDPROCEDURE ;
    OV 3 4 ; L := 1 ;
    V1 := .1 ; V2 := -.2 ; V3 := .1 ;
    WHILE V2 ;
      FIT V1 V3 ;
```

```
    UM ; TRIPLET L V1 V2 V3 ;
    OBJ := ABS(ME(1,2)) + ABS(ME(3,4))
  ENDFIT OBJ 1E-12 1 ;

  UM ; LOOP I 1 10 ; TRIPLET L V1 V2 V3 ; ENDLOOP ; PM 6 ;
  WRITE 6 V2 L ; READ 6 V2 ; READ 6 L ;
ENDWHILE ;
```

This example shows how it is possible to phrase more complicated interactive optimization tasks in the COSY language. To conclude our short list of examples, we want to show how a user can define his own particle optical element and work with it. For reasons of simplicity, we choose an optical lens with focal length F. We do not consider spherical aberrations, but want to correct the chromatic effects. To describe the action of the lens, we follow the DA prescription to just perform ray trace operations using DA objects [8, 9] and make use of the global DA variable MAP which contains the momentary derivatives of the transfer map.

```
PROCEDURE RUN ;
  VARIABLE S 1 ;
  PROCEDURE LENS F D ; {lets a lens with reference focal length
              F made of glass with dispersion D act on the map}
    MAP(2) := MAP(2) - MAP(1)/F*(1+D*DA(5)) ;
    MAP(4) := MAP(4) - MAP(3)/F*(1+D*DA(5)) ;
  ENDPROCEDURE ;

  OV 2 5 ;
  S := -1 ;
  FIT S ;
    UM ;   LENS 1 .01 ;    DL .01 ;    LENS S .005 ;
    ENDFIT ABS(ME(1,25)) 1E-12 1 ;
  PM 6 ;
ENDPROCEDURE ; RUN ; END ;
```

This concludes our short list of examples. It should be quite obvious how much more complicated problems can be treated.

9 Further Development

We hope that COSY INFINITY philosophy will eventually make it possible to incorporate new features developed by individual users into the master version. But until then, we will continue upgrading the features of the optics environment, and plan to have a new release of the program several times per year.

Besides, there are still fundamental things that have to be provided shortly. First and foremost, this includes graphics. Being helpful and essential for design and study of beamlines and accelerators, graphics is by far not as standardized as FORTRAN, the language in which COSY INFINITY is written. To account for this fact, all graphics will be output to an ASCII graphics metafile containing only move and draw commands as well as output of single characters.

This metafile can then be interpreted within any graphics environment with ease. Note that this approach is not necessarily restricted to a viewing of the graphics after everything is done. Since it is possible to initiate operating system commands from within COSY INFINITY, it is possible to initiate the graphics post-processing from the program, perhaps in a different window.

Another useful but highly non portable feature is the input of reals through pointing devices. We are very inclined to develop a tool for such purposes.

Finally, as far as the physics aspects are concerned, advanced techniques for symplectic tracking using maps are under development. While it is simple to build a tracking environment based on symplectic kicks in COSY INFINITY, gains in speed combined with the much more precise description of the optics including thick elements and fringe fields, makes the tracking with maps very attractive.

Acknowledgements

This work was supported by the Director, Office of Energy Research, Office of Basic Energy Sciences, Materials Sciences Division, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

. For discussions about languages and compilers, I want to thank Dr. Hiroshi Nishimura, Dipl.-Math. Dipl.-Phys. H. C. Hofmann, and Dipl.- Phys. Klaus Lindemann. For discussions about optimizers, I would like to thank Dr. Tom Mottershead. For testing various parts of the DA package, I would like to thank Dipl.- Phys. Bernd Hartmann and Dipl.-Phys. Stefan Meuser.

Financial support was appreciated from the german Bundesministerium für Forschung und Technologie, the Deutsche Forschungsgemeinschaft, Los Alamos National Laboratory, the University of Gießen, the SSC Central Design Group, and Lawrence Berkeley Laboratory.

Finally, I would like to thank my wife for letting me go on all those evenings and weekends ...

References

[1] M. Berz. Computational aspects of design and simulation: COSY INFINITY. *Nuclear Instruments and Methods*, in print, 1990.

[2] M. Berz. COSY INFINITY, an arbitrary order general purpose optics code. *AIP Conference Proceedings*, in print, 1990.

[3] K. L. Brown. The ion optical program TRANSPORT. Technical Report 91, SLAC, 1979.

[4] T. Matsuo and H. Matsuda. Computer program TRIO for third order calculations of ion trajectories. *Mass Spectrometry*, 24, 1976.

[5] H. Wollnik, J. Brezina, and M. Berz. GIOS-BEAMTRACE, a computer code for the design of ion optical systems including linear or nonlinear space charge. *Nuclear Instruments and Methods*, A258:408, 1987.

[6] M. Berz, H. C. Hofmann, and H. Wollnik. COSY 5.0, the fifth order code for corpuscular optical systems. *Nuclear Instruments and Methods*, A258:402, 1987.

[7] M. Berz and H. Wollnik. The program HAMILTON for the analytic solution of the equations of motion in particle optical systems through fifth order. *Nuclear Instruments and Methods*, A258:364, 1987.

[8] M. Berz. Arbitrary order description of arbitrary particle optical systems. *Nuclear Instruments and Methods*, in print, 1990.

[9] M. Berz. Differential Algebraic description of beam dynamics to very high orders. *Particle Accelerators*, 24:109, 1989.

[10] M. Berz. Differential Algebraic treatment of beam dynamics to very high orders including applications to spacecharge. *AIP Conference Proceedings*, 177:275, 1988.

[11] M. Berz. *The Description of Particle Accelerators using High Order Perturbation Theory on Maps, in: M. Month (Ed), Physics of Particle Accelerators*, volume 1, page 961. American Institute of Physics, 1989.

[12] M. Berz. High-order description of accelerators using differential algebra and first applications to the SSC. In *Proceedings Snowmass Summer Meeting*, Snowmass, Co, 1988.

[13] E. Forest and M. Berz. Canonical integration and analysis of periodic maps using Non-Standard Analysis and Lie methods. In *Proceedings Second Workshop on Lie Methods in Optics, in print*, Mexico City, 1988.

[14] M. Berz. The Differential Algebra FORTRAN precompiler DAFOR. Technical Report AT-3:TN-87-32, Los Alamos National Laboratory, 1987.

[15] M. Berz. The DA precompiler DAFOR. Technical report, Lawrence Berkeley Laboratory, Berkeley, Ca, 1990.

[16] E. Forest, M. Berz, and J. Irwin. Normal form methods for complicated periodic systems: A complete solution using Differential Algebra and Lie operators. *Particle Accelerators*, 24:91, 1989.

[17] H. Wollnik, B. Hartmann, and M. Berz. Principles behind GIOS and COSY. *AIP Conference Proceedings*, 177:74, 1988.

[18] B. Hartmann, M. Berz, and H. Wollnik. The computation of fringing fields using Differential Algebra. *Nuclear Instruments and Methods*, in print, 1989.

[19] B. Hartmann, H. Wollnik, and M.Berz. Tribo, a program to determine high-order properties of intense ion beams. *AIP Conference Proceedings*, in print, 1990.

[20] H. Wollnik, J. Brezina, and M. Berz. GIOS-BEAMTRACE, a program for the design of high resolution mass spectrometers. In *Proceedings AMCO-7*, Darmstadt, 1984.

[21] A. J. Dragt, L. M. Healy, F. Neri, and R. Ryne. MARYLIE 3.0 - a program for nonlinear analysis of accelerators and beamlines. *IEEE Transactions on Nuclear Science*, Ns-3,5:2311, 1985.

[22] C. Iselin and J. Niederer. The MAD program, version 7.2, user's reference manual. Technical Report CERN/LEP-TH/88-38, CERN, 1988.

[23] H. Wollnik. *Charged Particle Optics.* Academic Press, Orlando, Florida, 1987.

[24] D. C. Carey. *The Optics of Charged Particle Beams.* Harwood, 1987.

[25] K. G. Steffen. *High Energy Beam Optics.* Wiley-Interscience, New York, 1965.

[26] M. Born and E. Wolf. *Optics.*

10 Appendix: Momentary Types and Operations

Here we will list all objects as well as all the operands available for various combinations of objects, the available intrinsic functions, and the available intrinsic procedures. This information is contained in the language definition file GENFOX.DAT, which is read by the program FOXTYP.FOR which updates the compiler for the COSY language. This information is current as of 28-JUN-90. In this version, the following types are supported:

RE	8 Byte Real Number
ST	String
LO	Logical
DA	Differential Algebra Vector
VE	Real Number Vector
CM	8 Byte Complex Number
IN	8 Byte Interval Number

Now follows a list of all operations available for various combinations of types. For each operation, a relative priority is given which determines the hierarchy of the operations in expressions. One can override can these hierarchies with parentheses.

- + (Addition) has priority 3 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	RE
RE	DA	DA
RE	VE	VE
RE	CM	CM
RE	IN	IN
LO	LO	LO
DA	RE	DA
DA	DA	DA
VE	RE	VE
VE	VE	VE
CM	RE	CM
CM	CM	CM
IN	RE	IN
IN	IN	IN

- \- (Subtraction) has priority 3 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	RE
RE	DA	DA
RE	VE	VE
RE	CM	CM
RE	IN	IN
DA	RE	DA
DA	DA	DA
VE	RE	VE
VE	VE	VE
CM	RE	CM
CM	CM	CM
IN	RE	IN
IN	IN	IN

- * (Multiplication) has priority 4 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	RE
RE	DA	DA
RE	VE	VE
RE	CM	CM
RE	IN	IN
LO	LO	LO
DA	RE	DA
DA	DA	DA
VE	RE	VE
VE	VE	VE
CM	RE	CM
CM	CM	CM
IN	RE	IN
IN	IN	IN

- / (Division) has priority 4 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	RE
RE	DA	DA
RE	VE	VE
RE	CM	CM
RE	IN	IN
DA	RE	DA
DA	DA	DA
VE	RE	VE
VE	VE	VE
CM	RE	CM
CM	CM	CM
IN	RE	IN
IN	IN	IN

- ^ (Exponentiation) has priority 5 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	RE

- < (Less Than) has priority 2 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	LO

- > (Greater Than) has priority 2 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	LO

- = (Equal) has priority 2 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	LO

- # (Not Equal) has priority 2 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	LO

- & (Concatenate) has priority 2 and is supported for the following combinations of types:

Left Type	Right Type	Type of Result
RE	RE	VE
RE	VE	VE
ST	ST	ST
VE	RE	VE
VE	VE	VE

Next we list all intrinsic functions available in the momentary version with a short description and the allowed types.

- TYPE returns the type of an object as a number and is supported for the following types:

Argument Type	Type of Result
RE	RE
ST	RE
LO	RE
DA	RE
VE	RE
CM	RE
IN	IN

- LENGTH returns the momentary length of a variable in 8 byte blocks and is supported for the following types:

Argument Type	Type of Result
RE	RE
ST	RE
LO	RE
DA	RE
VE	RE
CM	RE
IN	IN

- VARMEM returns the current memory address of an object and is supported for the following types:

Argument Type	Type of Result
RE	RE
ST	RE
LO	RE
DA	RE
VE	RE
CM	RE

- VARPOI returns the current pointer address of an object and is supported for the following types:

Argument Type	Type of Result
RE	RE
ST	RE
LO	RE
DA	RE
VE	RE
CM	RE

- EXP computes the exponential function and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- LOG computes the natural logarithm and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- SIN computes the sine and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- COS computes the cosine and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- TAN computes the tangent and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- ASIN computes the arc sine and is supported for the following types:

Argument Type	Type of Result
RE	RE

- ACOS computes the arc cosine and is supported for the following types:

Argument Type	Type of Result
RE	RE

- ATAN computes the arc tangent and is supported for the following types:

Argument Type	Type of Result
RE	RE

- SINH computes the hyperbolic sine and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- COSH computes the hyperbolic cosine and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- TANH computes the hyperbolic tangent and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- SQRT computes the square root and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- ISRT computes the reciprocal of square root and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- SQR computes the square and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	DA

- ABS computes the absolute value and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	RE

- NORM computes the norm of a vector and is supported for the following types:

Argument Type	Type of Result
DA	RE

- REAL determines the real part and is supported for the following types:

Argument Type	Type of Result
RE	RE
DA	RE
CM	RE
IN	RE

- INT determines the integer part and is supported for the following types:

Argument Type	Type of Result
RE	RE

- NINT determines the nearest integer and is supported for the following types:

Argument Type	Type of Result
RE	RE

- DA returns the i th elementary DA vector and is supported for the following types:

Argument Type	Type of Result
RE	DA

- IMAG extracts the imaginary part and is supported for the following types:

Argument Type	Type of Result
CM	RE

- CONJ conjugates a complex number and is supported for the following types:

Argument Type	Type of Result
CM	CM

In addition to the just listed operators and intrinsic functions, the following intrinsic procedures are available:

- Procedure MEMALL (1 argument) returns the amount of memory allocated at this time
- Procedure MEMFRE (1 argument) returns the amount of memory still available at this time

- Procedure DAINI (4 arguments) initializes the order and number of variables of DA. Arguments are order, number of variables, output unit number, number of monomials (on return).

- Procedure DANOT (1 argument) sets momentary truncation order for DA.

- Procedure DAEPS (1 argument) sets zero tolerance for components of DA vectors.

- Procedure DAPRV (3 arguments) prints an array of DA vectors. Arguments are the array, the number of components, and the unit number.

- Procedure DAPOI (4 arguments) computes the POISSON bracket between DA vectors. Arguments are the two vectors, the dimensionality of Hamiltonian space, and the result.

- Procedure DALEX (4 arguments) computes the action c = exp(:b:)a. Arguments are a,b,c and the dimensionality of Hamiltonian space.

- Procedure DAGMD (4 arguments) computes grad(v1) * v2, where v1 is a DA vector, v2 is an array of DA vectors. Arguments are v1, v2, the result and dimension of v2.

- Procedure DAFLO (4 arguments) computes the flow of x' = f(x) for 1 time unit. Arguments are the initial condition, array of right hand sides, result, and dim of f.

- Procedure DARAN (2 arguments) fills DA vector with random entries. Arguments are DA vector and fill factor.

- Procedure DADER (3 arguments) performs the derivation operation on DA vector. Arguments are the number with respect to which to differentiate and the two DA vectors.

- Procedure DAINT (3 arguments) performs an integration of a DA vector. Arguments are the number with respect to which to integrate and the two DA vectors.

- Procedure DAPEE (3 arguments) returns a component of a DA vector. Arguments are the DA vector, the ID for the coefficient in TRANSPORT notation, and the real component.
- Procedure DACCT (6 arguments) concatenates 2 DA arrays. Arguments are the first array, number of elements, second array, number of elements, third array, number of elements.
- Procedure SUBSTR (4 arguments) computes a substring. Arguments are string, first and last numbers identifying substring, and substring.
- Procedure VECELE (3 arguments) returns a component of a vector of reals. Arguments are the vector, the number of the component, and on return the real value of the component.
- Procedure IMUNIT (1 argument) returns the imaginary unit i.
- Procedure TRUE (1 argument) returns the logical value true.
- Procedure FALSE (1 argument) returns the logical value false.
- Procedure INTERV (3 arguments) produces an interval from 2 numbers. Arguments are the lower and upper bounds and on return the resulting interval.
- Procedure INLO (2 arguments) returns the lower bound of an interval. Arguments are the interval and on return the lower bound.
- Procedure INUP (2 arguments) returns the upper bound of an interval. Arguments are the interval and on return the lower bound.

LAWRENCE BERKELEY LABORATORY
UNIVERSITY OF CALIFORNIA
INFORMATION RESOURCES DEPARTMENT
BERKELEY, CALIFORNIA 94720