COSY INFINITY
Version 8.1
Programming Manual [*]

M. Berz, J. Hoefkens[†] and K. Makino[‡]
Department of Physics and Astronomy
Michigan State University
East Lansing, MI 48824

May, 2001
Revised in October, 2002

Abstract

This is a programming manual for the arbitrary order code COSY INFINITY and the C++ and Fortran 90 interfaces to it. It is current as of October 13, 2002. COSY INFINITY is a powerful object oriented with dynamic typing which is using sophisticated differential algebraic (DA) methods that allow high order sensitivity studies and verification with advanced data types like Taylor models.

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 analysis of numerical problems. Moreover, C++ classes and Fortran 90 modules allow the inclusion of COSY INFINITY into the respective environments. These interface classes/modules outperform similar attempts of creating differential algebraic packages in these languages by a wide margin.

Altogether, the uniqueness of COSY lies in the ability to handle high order maps with and without verification. Furthermore, its powerful work environment adopts to many numerical problems and is quite flexible and portable through the availability of C++ and Fortran 90 interfaces to its sophisticated algorithms and data structures.

[*]Supported in part by the U.S. Department of Energy and the Alfred P. Sloan Foundation.

[†]Currently at Gene Data AG, Basel, Switzerland (jens@hoefkens.com).

[‡]Currently at University of Illinois at Urbana-Champaign, Illinois, USA (makino@uiuc.edu).

Contents

1 Before Using COSY INFINITY

1.1 User's Agreement

COSY INFINITY [1] can be obtained from M. Berz under the following conditions.

Users are requested not to make the code available to others, but ask them to obtain it from us. We maintain a list of users to be able to send out regular updates, which will also include features supplied by other users.

The Fortran portions and the high-level COSY language portions of the code should not be modified without our consent. This does not include the addition of new optimizers and new graphics drivers as discussed in section 5.2.2; however, we would like to receive copies of new routines for possible inclusion in the master version of the code.

Though we do our best to keep the code free of errors 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, we would like to see this acknowledged by referencing some of the papers related to the code, for example [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.

By using the code COSY INFINITY, users agree to be bound by the above conditions.

1.2 How to Obtain Help and to Give Feedback

While this manual is intended to describe the use of the code as completely as possible, there will probably arise questions that this manual cannot answer. Furthermore, we encourage users to contact us with any suggestions, criticism, praise, or other feedback they may have. We also appreciate receiving COSY source code for utilities users have written and find helpful.

We prefer to communicate by www or electronic mail. We can be contacted as follows:

Prof. Martin Berz
Department of Physics and Astronomy
Michigan State University
East Lansing, MI 48824, USA
Phone: 1-517-355-9200 ex.2130
FAX: 1-313-731-0313 (USA)

or 49-89-9218-5422 (Europe/Germany)
email: berz@msu.edu
http://cosy.pa.msu.edu/

1.3 How to Install the Code

The code for COSY INFINITY consists of the following files:

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

All the system files of COSY INFINITY are currently distributed via the WWW; http://cosy.pa.msu.edu/.

Four files, FOXY.FOP, DAFOX.FOP, FOXFIT.FOP and FOXGRAF.FOP, are written in Fortran and have to be compiled and linked. FOXY.FOP is the compiler and executer of the COSY language. DAFOX.FOP contains the routines to perform operations with objects, in particular the differential algebraic routines. FOXFIT.FOP contains the package of nonlinear optimizers. FOXGRAF.FOP contains the available graphics output drivers, which are listed in section 5.2.2.

In FOXGRAF.FOP, the PGPLOT graphics driver routines are contained as standard graphics output in COSY INFINITY. The PGPLOT graphics library is freely available from the web page http://astro.caltech.edu/∼tjp/pgplot/, and can be installed to VMS, UNIX, Windows 95/98/NT, etc (see also section 5.2.2). See page 8 for an example makefile for a Linux system. If not desired, the PGPLOT driver routines in FOXGRAF.FOP should be removed and replaced by the provided dummy routines. Some of the other popular graphics drivers, direct PostScript output and direct LaTeX output, are self contained in FOXGRAF.FOP and don't require to link to other libraries.

COSY.FOX contains all the physics of COSY INFINITY, and is written in COSY INFINITY's own input language. It has to be compiled by FOXY as part of the installation process.

All the Fortran parts of COSY INFINITY are written in standard ANSI Fortran 77. However, certain aspects of Fortran 77 are still system dependent; in particular, this concerns file handling. All system dependent features of COSY INFINITY are coded for various machines, including VAX/VMS, Windows PC, UNIX, Linux, HP, IBM mainframes, and CRAY (HP, IBM mainframes, CRAY are not actively maintained at this time).

The type of machine can be changed by selectively adding and removing comment identifiers from certain lines. To go from UNIX to VAX, for example, all lines that have the identifier *UNIX somewhere in columns 73 through 80 have to be commented, and all lines that have the comment *VAX in columns 1 through 5 have to be un-commented. To automate this process, there is a utility Fortran program called VERSION that performs all these changes automatically. Should there be additional problems, a short message to us would be appreciated in order to facilitate life for future users on the same system.

1.3.1 Standard UNIX systems

The Fortran source is by default compatible with standard UNIX systems. In general, the compiler optimization option is not recommended, because it sometimes causes trouble in handling the COSY syntax.

On SunOS/Solaris systems, compilation should be performed with the compiler option "`-Bstatic`".

If PGPLOT graphics is desired, the code has to be linked with the local PGPLOT libraries.

Currently as of October 13, 2002, the GKS graphics routines are commented out. If GKS graphics is desired, activate the GKS routines in foxgraf.f using the small program VERSION as described on page 32. The code has to be linked to the local GKS object code. On workstations, the graphics can be utilized under Xwindows and Tektronix.

1.3.2 VAX/Open VMS systems

On VAX/Open VMS systems, all lines that contain the string *VAX in columns 1 to 5 should be un-commented, and all the lines containing the string *UNIX in columns 73 to 80 should be commented. This can be done using the small program VERSION; at first, adjust VERSION manually so it performs file handling properly.

Compilation should be done without any options. In order to link and run the code, it may be necessary to increase certain working set parameters. The following parameters, generated with the VAX/Open VMS command SHOW WORK, are sufficient:

```
Working Set (pagelets)  /Limit=1408  /Quota=10240  /Extent=128000
Adjustment enabled      Authorized Quota=10240  Authorized Extent=128000

Working Set (8Kb pages) /Limit=88  /Quota=640  /Extent=8000
```

```
                    Authorized Quota=640  Authorized Extent=8000
```

If PGPLOT graphics is desired, the code has to be linked with the local PGPLOT libraries.

Currently as of October 13, 2002, the GKS graphics routines are commented out. If GKS graphics is desired, activate the GKS routines in FOXGRAF.FOP using the small program VERSION as described on page 32. The code has to be linked with the local GKS object code. It can be executed on workstations with UIS graphics, with Xwindows graphics, and on terminals supporting Tektronix.

1.3.3 Windows PC

An executable program for Microsoft Windows 95/98/NT by the DIGITAL Visual Fortran compiler 5.0 linked with the PGPLOT graphics libraries is available.

In case compilation and linking on local machines are needed, the four Fortran source files have to be adjusted; all lines that contain the string *PC in columns 1 to 3 should be un-commented, and all the lines containing the string *UNIX in columns 73 to 80 should be commented. This can be done using the small program VERSION; at first, adjust VERSION manually so it performs file handling properly.

If PGPLOT graphics is desired, the code has to be linked with the local PGPLOT libraries.

If VGA graphics packages with Lahey F77/F90 compilers are desired, FOXGRAF.FOP has to be adjusted; see section 5.2.2.

1.3.4 G77 systems (Linux)

On systems that use the GNU Fortran 77 compiler g77 and the appropriate GNU libraries, all lines that contain the string *G77 in columns 1 to 4 should be un-commented, and all the lines containing the string *UNIX in columns 73 to 80 should be commented. This can be done using the small program VERSION; at first, adjust VERSION manually so it performs file handling properly.

The following is an example “Makefile” to compile and link the program with the PGPLOT graphics libraries. Check the documentation of the GNU Fortran 77 compiler about platform specific options. In general, the compiler optimization option is not recommended, because it sometimes causes trouble in handling the COSY syntax.

```
FC=g77 -Wall
FFLAGS=
LIBS=-L/usr/local/pgplot -lpgplot -L/usr/X11R6/lib -lX11
```

```
OBJ = dafox.o foxy.o foxfit.o foxgraf.o

all: $(OBJ)
        $(FC) -o cosy $(OBJ) $(LIBS)
```

1.3.5 HP systems

On HP systems, all lines that contain the string *HP in columns 1 to 3 should be un-commented, and all the lines containing the string *UNIX in columns 73 to 80 should be commented. This can be done using the small program VERSION; at first, adjust VERSION manually so it performs file handling properly.

Compilation should be performed with the compiler option setting static memory handling.

If PGPLOT graphics is desired, the code has to be linked with the local PGPLOT libraries.

Currently as of October 13, 2002, the GKS graphics routines are commented out. If GKS graphics is desired, activate the GKS routines in foxgraf.f using the small program VERSION as described on page 32. The code should be linked to the local GKS object code. GKS on HP systems usually requires the use of INCLUDE files in the beginning of FOXGRAF.FOP as well as in all subroutines. These INCLUDE statements are contained in the HP version, but they have to be moved from column 6 to column 1, and possibly the address of the libraries has to be changed.On workstations, the graphics can be utilized under Xwindows and Tektronix.

The last versions of COSY INFINITY have not been explicitly tested on HP systems. Additional changes may be necessary.

1.3.6 IBM Mainframes

On IBM mainframe systems, all lines that contain the string *IBM in columns 1 to 4 should be un-commented, and all the lines containing the string *UNIX in columns 73 to 80 should be commented. This can be done using the small program VERSION; at first, adjust VERSION manually so it performs file handling properly.

The last versions of COSY INFINITY have not been explicitly tested on IBM Mainframes. Additional changes may be necessary.

1.3.7 CRAY

The installation to CRAY machines with UNIX operating systems should follow the instruction in subsection 1.3.1 on Standard UNIX systems.

On CRAY machines with the original CRAY operating systems, all lines that contain the string *CRAY in columns 1 to 5 should be un-commented, and all the lines containing the string *UNIX in columns 73 to 80 should be commented. This can be done using the small program VERSION; at first, adjust VERSION manually so it performs file handling properly.

The last versions of COSY INFINITY have not been explicitly tested on CRAYs. Additional changes may be necessary.

1.3.8 Possible Memory Limitations

Being based on Fortran, which does not allow dynamic memory allocation, COSY INFINITY has its own memory management within a large Fortran COMMON block. On machines supporting virtual memory, the size of this block should not present any problem. On some other machines, it may be necessary to scale down the length. This can be achieved by changing the parameter LMEM at all occurrences in FOXY.FOP, DAFOX.FOP and FOXGRAF.FOP to a lower value. Values of around 500 000 should be enough for many applications, which brings total system memory down to about 8 Megabytes.

In the case of limited memory resources, it may also 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 MSC 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.

1.3.9 Syntax Changes

With very minor exceptions, version 8 and version 8.1 are downward compatible to the previously released version 7 of COSY INFINITY. Any user deck for version 7 should run under versions 8 and 8.1.

As of June 29 2002, the syntax of RDVAR is changed. Refer to pages 17 and 79.

As of October 12 2002, the GKS graphics driver routines in FOXGRAF.FOP are commented out. When the GKS graphics library is not linked, the user does not have to change FOXGRAF.FOP. The data types for ordered interval (OI) and ordered interval vectors (OV) are no longer supported.

2 What is COSY INFINITY

COSY INFINITY has been developed as one of the leading tools for the simulation of Beam Physical problems. Beam Physics is concerned with the design and the analysis of particle accelerators which are among the largest and most expensive scientific experiments ever build.

2.1 COSY's Algorithms and their Implementation

Using 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 [3, 4, 5, 6, 7]. An offspring of this approach is the computation of maps for large accelerators where often the system can be described by inexpensive, low order kick integrators.

The speed of this approach is initially determined by the numerical integration process. Using DA techniques, this problem can be overcome too: DA can be used to automatically generate numerical integrators of arbitrary high orders in the time step, yet at the computational expense of only little more than a first order integrator [6, 7]. This technique is very versatile, works for a very large class of fields, and the speeds obtained are similar to classical 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 within the language itself. In the conventional languages used for numerical applications (namely C and Fortran) it is often difficult to introduce new data types and overload the operations on them. Modern object oriented languages like C++ and Java on the other hand have the capabilities of conveniently introducing new data types. However, the added flexibility often comes with a hefty performance penalty that limits the applicability of these languages to complicated numerical problems.

Hence, there are strong reasons to stay within the limits of a Fortran environment. Firstly, almost all software in the field of scientific computing is written in this language, and the desire to interface to such software is easier if Fortran is used. Furthermore, there are extensive libraries of support software which are only slowly becoming available in other languages, including routines for nonlinear optimization and numerical toolboxes. 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 to the supercomputers, has a Fortran compiler. Moreover, due to the long experience with them, these compilers are very mature and produce highly optimized and efficient code.

Consequently, the DA precompiler [8] has been designed in Fortran 77. 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. While the DA precompiler is not a full Automatic Differentiation tool like Adifor [9] or Odyssee, it has been extensively used [7]. 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.

However, with the recently developed C++ and Fortran 90 interfaces to COSY INFINITY, the question of the underlying software architecture has become somewhat obsolete. It is now possible to access the sophisticated data structures and algorithms of COSY INFINITY even from these languages. Moreover, these native-language interfaces to COSY INFINITY outperform similar attempts at creating differential algebraic data types by a wide margin. The performance of the interfaces is within a factor of two to the regular COSY system on most platforms (detailed performance comparisons will be published elsewhere). It should however be stressed that these interfaces do not provide access to the tools required for studying Beam Physics. More information on the C++ interface is given in section 7. Details on the Fortran 90 module can be found in 8.

2.2 The User Interface

COSY INFINITY approaches the problem of defining a suitable interface to its internal data types by offering the user a full programming language; in fact, the language is so powerful that all the physics of the COSY INFINITY system was written in it.

For ease of use, this language has a deliberately simple syntax. For the user demanding special-purpose features on the other hand, it should be powerful. It should allow direct and complex interfacing to Fortran routines, and it should allow the use of DA as a built-in type. Finally, it should be widely portable. Unfortunately, there is no language readily available that fulfills all these requirements, so COSY INFINITY contains its own language system.

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 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 relatively machine independent, the output of the compilation is not native machine code but rather an intermediate byte code that is interpreted in a second pass. In this respect, the concepts of COSY INFINITY are quite similar to the Java programming language. However, the COSY INFINITY system has the compiler and the executer combined into one single program. The byte code used by COSY INFINITY is portable between machines of the same word size. To match the portability of the system on the platform dependent parts, it is essential to write the source code of the compiler in a very portable language. We chose Fortran for the compiler, even though clearly it is considerably easier to write it in a recursive language like C.

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 written 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.

3 Data Types

This section should be read together with Appendix A, which lists the elementary operations, procedures, and functions defined for COSY objects.

COSY INFINITY is an environment with dynamic typing. Thus, a single function can be evaluated with different argument of different types and the actual type of the operations is determined at run time. In other words, the compiler does not generate several copies of the same function for different arguments but determines the appropriate operations at run time. Nevertheless, it is often desirable to specifically create variables of a certain type. In this section, we will discuss the corresponding COSY functions and procedures that allow the explicit initialization of COSY variables.

3.1 Real Numbers

Real number variables are created by assignment. Initially, all variables are of type **RE** and are initialized to 0. Thus, the following fragment declares two variable `X` and `Y` with enough space for a single double precision number and initializes them to 1 and $1/e^3$, respectively.

```
VARIABLE X 1 ; VARIABLE Y 1 ;
X := 1 ;
Y := EXP(-3) ;
```

Details on the allowed operations and their return types for real variables can be found in Appendix A.

3.2 Strings

Strings can be created either by assignment or by concatenation of other strings. As an example, consider the following code fragment:

```
VARIABLE S 80 ; VARIABLE T 80 2 ;
T(1) := 'HELLO ' ;
T(2) := 'WORLD' ;
S := T(1)&T(2) ;
```

It creates two string variables by assignment and initializes the variable `S` by assigning the union of the two variables `T(1)` and `T(2)`. Other procedures operating on strings are described in Appendix A.

3.3 Logicals

Logical variables can be created by assignment or with the procedures **LTRUE** and **LFALSE** described in Appendix A. The following code fragments illustrates this:

```
VARIABLE L 1 ;

L := 1=1 ;
L := 1=2 ;

LTRUE L ;
LFALSE L ;
```

Note that logical values can be stored in variables of any size. Appendix A describes the operations and functions defined for logical values.

3.4 Complex Numbers

Complex numbers are created with the help of the COSY procedure **IMUNIT** or the convenience function **IM**. The following two fragments each create a variable `Z` and initialize it to $z = 2 - 3i$. Note that the variable `Z` has to be declared with enough space to hold two double precision numbers.

```
VARIABLE Z 2 ;
IMUNIT Z ;
Z := 2 - 3*Z ;
```

or

```
VARIABLE Z 2 ;
Z := 2 - IM(3) ;
```

Once initialized, complex numbers can be used in most mathematical expressions and evaluations (refer to Appendix A for details).

3.5 Intervals

Interval variables are created by using the COSY procedure **INTERV** or the the convenience function **INTV**. The following two fragments each create a variable `I` and initialize it to the interval $[-2, 3]$. Note that the variable `I` has to be declared with enough space to hold two double precision numbers.

```
VARIABLE I 2 ;
INTERV -2 3 I ;
```

or

```
VARIABLE I 2 ;
I := INTV(-2,3) ;
```

Once initialized, intervals can be used in most mathematical expressions and evaluations (refer to Appendix A for details). COSY INFINITY supports proper outward rounding for guaranteed interval enclosures. However, this can be disabled with the COSY procedure **INSRND**. Extra caution has to be used for disabling the outward rounding, knowing that it will void the validated enclosure computation.

3.6 Vectors

COSY INFINITY has vector data types that are similar to one-dimensional arrays, but differ in that elementary operations and functions are defined on them (generally, the operations act component-wise).

Several different vector types exist, distinguished by the type of the components. All vectors are created with the concatenation operator & and utility functions exist to extract components. The following fragments demonstrate the creation of a real number vector and an interval vector, respectively.

```
VARIABLE V 4 ;
V := 2&3 ;
V := V&V ;
```

and

```
VARIABLE V 8 ; VARIABLE I 2 ;
INTERV 2 3 I ;
V := I&(2*I) ;
V := V&V/3 ;
```

More details on the operations and functions defined on the various vector data types are given in Appendix A.

3.7 DA Vectors

DA vectors (or Taylor polynomials) can be created in several ways. It is important to distinguish DA vectors from the usual vector data types: besides the name, DA vectors do not share any of the characteristics of the normal COSY INFINITY vector data types.

DA vectors can be created by evaluating expressions with the return values of the **DA** function. Use of DA vectors requires prior initialization of the DA system of COSY INFINITY by using the procedures **DAINI** or **OV**. As an example of creating a DA vector, consider the following code fragment. It initializes the DA system to order three in two variables and assigns the third-order Taylor expansion of $x_1 \cdot \exp(x_1 + x_2)$ around the origin to the variable `D`.

```
VARIABLE D 100 ; VARIABLE NM 1 ;
DAINI 3 2 0 NM ;
D := DA(1)*EXP(DA(1)+DA(2)) ;
```

More details on the operations and functions defined for DA vectors are given in Appendix A.

3.8 Taylor Models, RDA Objects

Taylor model variables [10, 11, 12] should be created evaluating expressions with elementary Taylor models. The latter can be created with the intrinsic procedure **RDVAR** (Refer to page 79). Like in the case of DA vectors, use of Taylor models requires prior initialization of the DA system. The following fragment creates a 10-th order Taylor model for $f(x_1, x_2) = x_1 \cdot \exp(x_1 + x_2)$, defined over the domain $[-1, 1]$ with a reference point of $(0, 0)$:

```
VARIABLE D 1000 ; VARIABLE NM 1 ;
VARIABLE X1 100 ; VARIABLE X2 100 ; VARIABLE I 2 ;
VARIABLE IMD 1 ;

DAINI 10 2 0 NM ;
INTERV -1 1 I ;
IMD := 1 ;
RDVAR 1 0&0 I&I IMD X1 ; RDVAR 2 0&0 I&I IMD X2 ;
D := X1*EXP(X1+X2) ;
```

More details on the operations and functions defined for Taylor models are given in Appendix A (Taylor models are listed as *Remainder-enhanced Differential Algebra Objects*, RDA).

4 The COSY Language

4.1 General Aspects

In this section we will discuss the syntax of the COSY language. A brief summary of the commands can be found in section 4.6 on page 27. It will become apparent that the language has the flavor of PASCAL, which has a particularly simple syntax yet is relatively easy to analyze by a compiler and rather powerful.

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 a language description file described in the appendix. 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 "{" 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 underscore character "_". The case of the letters is not significant.

4.2 Program Segments and Structuring

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, of which there has to be exactly one and which has to begin at the top of the input file and ends at the end. It is denoted by 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. Procedures and functions must contain at least one executable 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. A real or double precision number number requires a length of 1, a complex double precision number a length of 2. A DA vector requires a length of at least the number of derivatives to be stored, and a CD vector requires twice that. The maximum number of derivatives through nth order in v variables can be obtained using the function **NMON**(n,v). Note that during allocation, the type and value of the variable is set to the real zero.

If the variable is to be used with indices as an array, the next expressions have to specify the different dimensions. Different elements of an array can have different types. This also allows to emulate most of the record concept found in PASCAL using arrays. As an example, the command

```
VARIABLE X 100 5 7 ;
```

declares X to be a two dimensional array with 5 respectively 7 entries, each of which has room for 100 memory locations.

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.

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 and grouped by parentheses, 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 are given in section A. 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.

Finally, function calls have the form

<**function name**> (<expression> { <, expression> }) ;

The name is the identifier of the procedure to be called which has to be visible at the current position. The arguments to be passed into the function are surrounded by parenthesis and separated by commas. The number of arguments has to match the number of arguments in the declaration of the function and the number of arguments has to be at least one.

There are many useful small **FUNCTION**s defined in the file cosy.fox, and it would be worthwhile to include its compiled code 'COSY.bin'. This is achieved by the following command in the beginning of the user program.

INCLUDE 'COSY' ;

Since cosy.fox, and its compiled code COSY.bin, contain

BEGIN ;

statement at the top of the program, the user program must not have **BEGIN** statement. See page 26 for the rules of **INCLUDE**.

Instead of using the **INCLUDE** statement, the same effect can be achieved by copying those small **FUNCTION**s in cosy.fox to paste to the beginning of user's programs.

4.3 Flow Control Statements

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** is evaluated. If it is not of logical type, an error message will be issued. If the value is true, execution will continue after the current line and until the next **ELSEIF**, at which point execution continues after the **ENDIF**.

If the value is false, the same procedure is followed with the logical expression in the next **ELSEIF**, until all of them have been reached, at which point execution continues after the **ENDIF**. At most one of the sections of code separated by **IF** and the matching optional **ELSEIF** and the **ENDIF** statements is executed.

There is nothing equivalent of a Fortran ELSE statement in the COSY language, but the same effect can be achieved with the statement ELSEIF 1=1 ;

The next such control structure consists of the pair

WHILE <expression> ;

and

ENDWHILE ;

If the expression is not of type logical, an error message will be issued. Otherwise, if it 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 the loop variable is not smaller 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. While it is allowed to alter the value of the loop variable inside the loop, this has no effect on the number of iterations (the loop variable is reset before the next iteration). Hence, it is not possible to terminate execution of a loop prematurely.

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> {<expression>} ;

Here the names denote the visible variables that are being adjusted. The first expression is the tolerance to which the minimum is requested. The second expression is the maximum number of evaluations of the objective function permitted. If this number is set to zero, no optimization is performed and the commands in the fit block are executed only once. The third expression gives the number of the optimizing algorithm that is being used. For the various optimizing algorithms, see section 5.1. The following expressions are of real or integer type and denote the objective quantities, the quantities that have to be minimized.

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 or the allowed number of iterations has been exhausted. After the optimization terminates, the variables contain the values corresponding to the lowest value of the objective quantity encountered by the algorithm.

Note that it is possible to terminate execution at any time by calling the intrinsic procedure **QUIT**. The procedure has one argument which determines if system information is provided. If this is not desired, the value 0 should be used.

4.4 Input and Output

The COSY language has provisions for fully formatted or unformatted I/O. All input and output is performed using the two fundamental routines

READ <expression> <name> ;

and

WRITE <expression> {<expression>} ;

The first expression stands for a unit number, where using common notation, unit 5 denotes the keyboard and unit 6 denotes the screen. Unit numbers can be associated with particular file names by using the **OPENF** and **CLOSEF** procedures, which can be found in the index.

It is also possible to have binary input and output. The binary input and output are limited to one real number or one Taylor model by one COSY statement. The syntax of real number binary input and output is similar to the syntax of **READ** and **WRITE**. Use **READB** and **WRITEB** instead.

READB <expression> <name> ;

WRITEB <expression> {<expression>} ;

The Taylor model binary input and output use the procedures **RDREAB** and **RDWRTB**. See the index entries for them.

Files for binary input and output have to be opened and closed by using the **OPENFB** and **CLOSEFB** procedures, and the syntax is similar to that of **OPENF** and **CLOSF**. See the index entries for them.

In the **READ** command, the name denotes the variable to be read. If the information that is read is a legal format free number, the variable will be of real type and contain the value of the number. In any other case, the variable will be of type string and contain the text just read.

For the case of formatted input, this string can be analyzed using the functions

SS (<string variable>,<I1>,<I2>)

which returns the substring from position I1 to position I2, as well as the function

R (<string variable>,<I1>,<I2>)

which converts the string representation of the real number contained in the substring from position I1 to I2 to the real number.

There are also dedicated read commands for other data types. For example, DA vectors can be read with the procedure **DAREA** (see index), and graphics meta files can be read with the procedure **GRREA** (see index). Taylor models can be read with the procedure **RDREA** (see index), and the binary input and output can be done with the procedures **RDREAB** and **RDWRTB** as mentioned above.

In the **WRITE** command, the expressions following the unit are the output quantities. Each quantity will be printed in a separate line. As described a few lines below, by using the utilities to convert reals or complex numbers or intervals to strings **SF** and **S** and the concatenation of strings, full formatted output is also possible.

Depending on the momentary type of the expression, the form of the output will be as follows.

Strings are printed character by character, if necessary over several lines with 132 characters per line, followed by a line feed.

Real numbers are printed in the Fortran format G23.16E3, followed by a line feed. Complex numbers will be printed in the form (R,I), where R and I are the real and imaginary parts which are printed in the Fortran format G17.9E3; the number is followed by a line feed.

Differential Algebraic numbers will be output in several lines. Each line contains the expansion coefficient, the order, and the exponents of the independent variables that describe the term. Vanishing coefficients are not printed. Complex Differential Algebraic variables are printed in a similar way, except instead of one real coefficient, the real and imaginary parts of the complex coefficient is shown. We note that it is also possible to print several DA vectors simultaneously such that the coefficients of each vector correspond to one column. This can be achieved with the intrinsic procedure **DAPRV** (see index) and is used for example for the output of transfer maps in the procedure **PM** (see index).

Taylor models will be output in several lines, too. In addition to the first part, which has the same format as Differential Algebraic numbers, the information about the reference point and the domain, and the remainder interval are output.

Vectors are printed component-wise such that five components appear per line in the format G14.7E3. As discussed above, this can be used to output several reals in one line.

Logicals are output as TRUE or FALSE followed by a line feed. Interval numbers are output in the form [L,U], where L and U are the outward rounded lower and upper bounds which are output as G23.16E3. If outward rounding output of intervals is not desired, use binary output. Graphics objects are output in the way described in section 5.2.

As described above, each quantity in the **WRITE** command is output in a new line. To obtain formatted output, there are utilities to convert real numbers to strings, several of which can be concatenated into one string and hence output in one line. The concatenation is performed with the string operator "`&`" described in section A. The conversion of a real number or a complex number pair or an interval to a string can be performed with the procedure **RECST** described in the appendix, as well as with the more convenient COSY function

SF (<real variable>,<format string>)

which returns the string representation of the real variable using the Fortran format specified in the format string. There is also a simplified version of this function

S (<real variable>)

which uses the Fortran format G23.16.

Both **SF** and **S** can be used for a complex number pair and an interval, too. In this case, the format string should specify only one Fortran number output format, which is applied to both numbers in the pair. Regardless the format specification, intervals are output with outward rounding.

Besides the input and output of variables at execution, there are also commands that allow to save and include code in compiled form. This allows later inclusion in another program without recompiling, and thus achieves a similar function as linking. The command

SAVE <name> ;

saves the compiled code in a file with the extension 'bin'; <name> is a string containing the name of root of the file, including paths and disks. The command

INCLUDE <name> ;

includes the previously compiled code. The name follows the same syntax as in the **SAVE** command.

Each code may contain only one **INCLUDE** statement, and it has to be located at the very top of the file. The **SAVE** and **INCLUDE** statements allow breaking the code into a chain of easily manageable pieces and decrease compilation times considerably.

4.5 Error Messages

COSY distinguishes between five different kinds of error messages which have different meanings and require different actions to correct the underlying problem. The five types of error messages are identified by the symbols `###`, `$$$`, `!!!`, `@@@` and `***`. In addition, there are informational messages, denoted by `---`. The meaning of the error messages is as follows:

`###`: This error 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. Circumstances under which it is issued include array bound violations, type violations, missing initialization of variables, exhaustion of the memory of a variable, and illegal operations such as division by zero.

`!!!`: This error 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 remedied by replacing the value of the parameter by a larger value and re-compiling. Note that all occurrences of the parameter in question have to be changed globally in *all* Fortran files.

`@@@`: This 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 user program.

`***`: This error message denotes errors in the use of COSY INFINITY library procedures. It includes messages about improper sequences and improper values for parameters.

In case execution cannot be continued successfully, a system error exit is produced by deliberately attempting to compute the square root of $-1.D0$. Depending on the system COSY is run on, this will produce information about the status at the time of error. In order to be system independent, this is done by attempting to execute the computation of the root of a negative number.

4.6 List of Keywords

As a summary, below follows a complete list of keywords of the COSY language.

BEGIN		**END**
VARIABLE		
PROCEDURE		**ENDPROCEDURE**
FUNCTION		**ENDFUNCTION**
IF	**ELSEIF**	**ENDIF**
WHILE		**ENDWHILE**
LOOP		**ENDLOOP**
FIT		**ENDFIT**
WRITE	**READ**	
SAVE	**INCLUDE**	

5 Optimization and Graphics

5.1 Optimization

Many problems in the design of particle optical systems require the use of nonlinear optimization algorithms. COSY INFINITY supports the use of nonlinear optimizers at its language level using the commands **FIT** and **ENDFIT** (see page 22). The optimizers for this purpose are given as Fortran subroutines. For a list of currently available optimizers, see section 5.1.1. Because of a relatively simple interface, it is also possible to include new optimizers relatively easily. Details can be found in section 5.1.2.

Besides the Fortran algorithms for nonlinear optimization, the COSY language allows the user to design his own optimization strategies depending on the problem. Some thoughts about problem dependent optimizers can be found in section 5.1.3.

5.1.1 Optimizers

The **FIT** and **ENDFIT** commands of COSY allow the use of various different optimizers supplied in Fortran to minimize an objective function that depends on several variables. For details on the syntax of the commands, we refer to page 22. The choice of the proper objective function is up to the user, and it sometimes influences the success or failure of the optimization attempt.

While many problems are directly minimizations of a single intuitive quantity, others often require the simultaneous satisfaction of a set of conditions. In this later case, it is clearly possible to construct a total objective function that has a minimum if and only if the conditions are satisfied; this can for example be achieved by writing the conditions in the form $f_i(\vec{x}) = 0$ and then forming the objective function as a sum of absolute values or a sum of squares. By using factors in front of the summands, it is possible to give more or less emphasis on certain conditions.

At the present time, COSY supports three different optimizers with different features and strengths and weaknesses to minimize such objective functions. In the following we present a list of the various optimizers with a short description of their strengths and weaknesses. Each number is followed by the optimizer it identifies.

1. The Simplex Algorithm
   This optimizer is suitable for rather general objective functions that do not have to satisfy any smoothness criteria. It is quite rugged and finds local (and often global) minima in a rather large class of cases. In simple cases, it requires more execution time than algorithms for almost linear functions. Because of its generality it is often the algorithm of choice.
2. Not available. Currently, it refers to "4. The LMDIF optimizer".

3. The Simulated Annealing Algorithm
   This algorithm is often able to find the total minimum for very complicated objective functions, especially in cases where all other optimizers fail. This comes at the expense of a very high and often prohibitive number of function evaluations. Often this algorithm is also helpful for finding starting values for the subsequent use of other algorithms.

4. The LMDIF optimizer
   This optimizer is a generalized least squares Newton method and is very fast if it works, but not as robust as the simplex algorithm. For most cases, it should be the first optimizer to try.

5.1.2 Adding an Optimizer

COSY INFINITY has a relatively simple interface that allows the addition of other Fortran optimizers. All optimizers that can be used in COSY must use "reverse communication". This means that the optimizer does not control the program flow, but rather acts as an oracle which is called repeatedly. Each time it returns a point and requests that the objective function be evaluated at this new point, after which the optimizer is to be called again. This continues until the optimum is found, at which time a control variable is set to a certain value.

All optimizers are interfaced to COSY INFINITY via the routine FIT at the beginning of the file FOXFIT.FOP, which is the routine that is called from the code executer in FOXY.FOP. The arguments for the routine are as follows:

IFIT	$\rightarrow$	identification number of optimizer
XV	$\leftrightarrow$	current array of variables
NV	$\rightarrow$	number of variables
EPS	$\rightarrow$	desired accuracy of function value
ITER	$\rightarrow$	maximum allowed iteration number
IEND	$\leftarrow$	status identifier

The last argument, the status identifier, communicates the status of the optimization process to the executer of COSY. As long as it is nonzero, the optimizer requests evaluation of the objective function at the returned point XV. If it is zero, the optimum has been found up to the abilities of the optimizer, and XV contains the point where the minimum occurs.

The subroutine FIT branches to the various supported optimizers according to the value IFIT. It also supplies the various parameters required by the local optimizers. To include a new optimizer merely requires to put another statement label into the computed GOTO statement and to call the routine with the proper parameters.

We note that when writing an optimizer for reverse communication, it is very important to have the optimizer remember the variables describing the optimization status from one call to the next. This can be achieved using the Fortran statement SAVE. If the optimizer can return at several different positions, it is also important to retain the information from where the return occurred.

In case the user interfaces an optimizer of his own into COSY, we would appreciate receiving a copy of the amended file FOXFIT.FOP in order to be able to distribute the optimizer to other users as well.

5.1.3 Problem Dependent Optimization Strategies

In the case of very complicated optimization tasks, the use of any optimization algorithm alone is often too restrictive and inefficient. In practice it is more often than not necessary to combine certain "hand-tuning" tasks with the use of optimizers or to just check the terrain by evaluating the objective function on a coarse multidimensional grid.

We want to point out that because of its language structure, COSY INFINITY gives the demanding user very far reaching freedom to tailor his own optimization strategy. This can be achieved by properly nested structures involving loops, while blocks or if blocks. Manual tuning can be performed by reading certain variables from the screen in a while loop and then printing other quantities of interest or even the system graphics to the screen.

Besides being powerful, these strategies have also proved quite economical. In most cases it is possible to reduce the number of required runs considerably by choosing appropriate combinations of interactive tuning and hard wired as well as self made optimization strategies. With some advance thought this at least in principle makes it possible to design even rather complicated systems with only one COSY run.

5.2 Graphics

The object oriented language on which COSY INFINITY is based supports graphics via the graphics object. This is used for all the graphics generated by COSY and allows a rather elegant generation and manipulation of pictures.

The operand "&" allows the merging of graphics objects, and COSY INFINITY has functions that return individual moves and draws and various other elementary operations which can be glued together with "&". For details, we refer to the appendix beginning on page 62.

5.2.1 Simple Pictures

There are a few utilities that facilitate the interactive generation of pictures. The following command generates a frame, coordinate system, title, and axis marks:

FG <PIC> <XL> <XR> <YB> <YT> <DX> <DY> <TITLE> <I> ;

where PIC is a variable that has to be allocated by the user and that will contain the frame after the call. XL, XR, YB, YT are the x coordinates of the left and right corners and the y coordinates of the bottom and top corners. DX and DY are the distances between axis ticks in x and y directions. TITLE is a string containing the title or any other text that is to be displayed. I=0 produces a frame with aspect ratio 1.5 to 1 which fills the whole picture, whereas I=1 produces a square frame.

There is also a procedure that allows drawing simple curves:

CG <PIC> <X> <Y> <N> ;

where PIC is again the variable containing the picture, and X and Y are arrays with N coordinates describing the corner of the polygon. Note that it is necessary to produce a frame with **FG** before calling this routine.

5.2.2 Supported Graphics Drivers

COSY INFINITY allows to output graphics objects with a variety of drivers which are addressed by different unit numbers. A graphics object is output like any other variable in the COSY language using COSY's **WRITE** command. The different unit numbers correspond to the following drivers:

- positive: Low-Resolution ASCII output to respective unit; 6: screen.
- -1: GKS-based output to primary VMS/UIS workstation window
- -2: GKS-based Tektronix output
- -3: GKS-based X-Windows workstation output
- -4: GKS-based postscript output to PICTURE.PS
- -5: GKS-based HP 7475 plotter output to file HP7475.PLT
- -6: GKS-based HP Paintjet / DEC JL250 to file HPPAINT.PLT
- -7: Direct LaTeX picture mode output to files lpic001.tex, lpic002.tex, ...
- -8: VGA graphics output with Lahey F77 compiler for DOS/Windows
- -9: Direct Tektronix output without external graphics library

- -10: Direct PostScript output to files pic001.ps, pic002.ps, ... The header of each PostScript file contains the information about how to convert to an Encapsulated PostScript (EPS) file and how to include the picture in a LaTeX document.
- -11: Direct output to the low level graphics meta file METAGRAF.DAT
- -12: graPHIGS-based X-Windows workstation output
- -31 ... -41: VGA graphics output with Lahey F90 compiler for DOS/Windows
- -51 ... -60: GKS-based secondary VMS/UIS workstation windows output
- -61 ... -70: GKS-based secondary X-Windows workstation windows output
- -71 ... -80: GKS-based secondary Tektronix output
- -81 ... -90: graPHIGS-based secondary X-Windows workstation windows output
- -101 ... -110: PGPLOT X-Windows workstation or Windows PC windows output
- -111: PGPLOT output to PostScript files pgpic001.ps, pgpic002.ps, ...
- -112: PGPLOT output to LaTeX files pgpic001.tex, pgpic002.tex, ...

Positive unit numbers produce a low resolution 80 column by 24 lines ASCII output of the picture written to the respective unit, where unit 6 again corresponds to the screen.

Note that the following units require linking to the specific graphics packages.

- -1 ... -6, -51 ... -71: GKS package
  As reported by many users, unfortunately GKS is not as standardized as desirable, and often adaptations to the local implementation may be necessary.
  The lines containing the graphics package specific description in FOXGRAF.FOP are commented out. Uncomment all the lines containing the string *GKS in columns 73 to 80 in FOXGRAF.FOP.
- -101 ... -112: PGPLOT package
  The PGPLOT Graphics Library is freely available from the PGPLOT web page, http://astro.caltech.edu/~tjp/pgplot/. Download and install the libraries according to the provided documentation on the target platform. Set the environment variables accordingly. A sample makefile on page 8 shows how to link to the PGPLOT libraries.
  If linking to PGPLOT package is not desired, the PGPLOT driver routines in FOXGRAF.FOP should be removed and replaced by the provided dummy routines.

- -8: VGA graphics package with Lahey F77 compiler for DOS/Windows
  The lines containing the graphics package specific description in FOXGRAF.FOP are commented out. Uncomment all the lines containing the string *L77 in columns 73 to 80 in FOXGRAF.FOP.
- -31 ... -41: VGA graphics package with Lahey F90 compiler for DOS/Windows
  The lines containing the graphics package specific description in FOXGRAF.FOP are commented out. Uncomment all the lines containing the string *L90 in columns 73 to 80 in FOXGRAF.FOP.
- -12, -81 ... -91: graPHIGS package
  The lines containing the graphics package specific description in FOXGRAF.FOP are commented out. Uncomment all the lines containing the string *PHG in columns 73 to 80 in FOXGRAF.FOP.

The other graphics drivers are self-contained within COSY.

Graphics written to a meta file can be read from a unit to a variable PIC with the command

GRREAD <unit> <PIC> ;

5.2.3 Adding Graphics Drivers

To facilitate the adaptation to new graphics packages, COSY INFINITY has a very simple standardized graphics interface in the file FOXGRAF.FOP. In order to write drivers for a new graphics package, the user has to supply a set of seven routines interfacing to the graphics package. For ease of identification and uniformity, the names of the routines should begin with a three letter identifier for the graphics system, and should end with three letters identifying their task. The required routines are

1. ...BEG : Begins the picture. Allows calling all routines necessary to initiate a picture.
2. ...MOV(X,Y) : Performs a move of the pen to coordinates X,Y. Coordinates range from 0 to 1.
3. ...DRA(X,Y) : Performs a draw from the current position to coordinates X,Y. Coordinates range from 0 to 1.
4. ...DOT(X,Y) : Performs a move of the pen to coordinates X,Y, then prints a dot at the position. Coordinates range from 0 to 1.
5. ...CHA(I) : Prints ASCII character I to momentary position. Size of the character has to be 1/80 of the total picture size. After the character, the position is advanced in X direction by 1/80.

6. ...COL(I) : Sets a color. If supported by the system, the colors are referred to by the following integers: 1: black, 2: blue, 3: red, 4: yellow, 5: green, 6: yellow/green, 7: sky-blue, 8: magenta, 9: navy, 10: background.
7. ...WID(I) : Sets the width of the pen. I ranges from 1 to 10, 1 denoting the finest and 10 the thickest line.
8. ...END : Concludes the picture. Allows calling all routines necessary to close the picture and print it.

The arguments X and Y are DOUBLE PRECISION, and I is INTEGER. After these routines have been created, the routine GRPRI in FOXGRAF.FOP has to be modified to include calls to the above routines at positions where the other corresponding routines are called for other graphics standards.

We appreciate receiving drivers for other graphics systems written by users to include them into the master version of the code.

5.2.4 The COSY Graphics Meta File

In case it is not desired to write driver routines at the Fortran level, it is possible to utilize the COSY graphics meta file, which is written in ASCII to the file METAGRAF.DAT via unit -11. This meta file can be easily read by standard Graphics programs such as TOPDRAWER, DISPLA, or GNUPLOT, or by programs written by users.

The meta file consists of a list of elementary operations discussed in the last subsection. Each of these seven elementary operations is output in a separate line, where the first three characters identify the command, then follows a blank, and then the parameters. The positions X and Y are output as 2E24.16, the character A as A1, and the numbers I as I1.

```
BEG
MOV X Y
DRA X Y
CHA A
COL I
WID I
END
```

6 Important Functions and Procedures

The COSY INFINITY language environment provides several powerful utility functions and procedures that aid users in developing software for COSY INFINITY. In this section, some of the most important such functions and procedures are discussed in further detail.

Functions discussed in this section are defined in the beginning of the file cosy.fox. Those **FUNCTION**s should be included in user programs either by

INCLUDE 'COSY' ;

statement or by copying them by ASCII editor to paste in the beginning of user programs. See pages 21 and 26.

6.1 Real Numbers

The following functions may be useful for the development of real number valued functions and procedures in COSY INFINITY.

MAX (<I>,<J>)

MIN (<I>,<J>)

These functions return the maximum and minimum of two numbers, respectively. The function

MOD (<I>,<J>)

returns the remainder of the arguments. Lastly, the function

SIG (<I>)

return the sign of its argument (it returns $+1$ of the constant part of I is greater or equal than zero and -1 otherwise.

6.2 Intervals

The most important utility functions for intervals are

INTV (<A>,<B>)

which returns an interval from the two bounds A and B. The functions

INL (<X>)

INU (<X>)

on the other hand extract the lower and upper bounds from the interval X.

6.3 DA Vectors and Taylor Models

The following functions can be used for easy access to the Differential Algebra package built into COSY INFINITY. The function

DER(<n>,<a>)

computes the DA derivation with respect to variable n. There is no corresponding function for Taylor models, since the latter cannot be differentiated without losing the basic inclusion properties. On the other hand, the functions

INTEG(<n>,<a>)

RINTEG(<n>,<a>)

compute the integral with respect to variable n. The first function acts on DA vectors, the second one acts on Taylor models. Finally, the function

PB (<a>,<b>)

computes the Poisson bracket between a and b. Another helpful function is

NMON (<NO>,<NV>)

which returns the maximum number of coefficients in a DA vector in NV variables to order NO. An important procedure for DA vectors and Taylor models is

POLVAL <L> <P> <NP> <A> <NA> <R> <NR> ;

which lets the polynomial described by the NP DA vectors or Taylor models stored in the array P act on the NA arguments A, and the result is stored in the NR Vectors R.

In the normal situation, L should be set 1. After **POLVAL** is called with L= 1, the analysis of the polynomial array P can be omitted by calling **POLVAL** with L= -1 or L= 0. The other setting for L is discouraged, because it may interfere with COSY's internal use of **POLVAL**.

The type of A is free, but all the array elements of A have to be the same type; it can be either DA, or CD, in which case the procedure acts as a concatenator, it can be real, complex or intervals, in which case it acts like a polynomial evaluator, or it can be of vector type VE, in which case it acts as a very efficient vectorizing map evaluator and is used for repetitive tracking. If necessary, adding `0*A(1)` can make the type of the argument array element `A(I)` agreeing to that type of `A(1)`.

In the case of Taylor model polynomials P, checks are made to ensure that the arguments are contained in the domains of the polynomials. Moreover, the results are represented by one of the verified data types (intervals, interval vectors, Taylor models) to ensure the correctness of the enclosures.

7 The C++ Interface

The COSY INFINITY language environment offers an object oriented approach to advanced numerical data types. However, access to these data types has traditionally required using the COSY INFINITY environment. This restriction has often made it difficult to interface COSY INFINITY's data types and algorithms with existing software packages (which are likely to be written in compiled languages like C++ and Fortran 90). The C++ interface to COSY INFINITY (and also the F90 interface discussed in §8) offers a solution to this problem: the flexibility of a modern object-oriented language combined with the power of the high performance data types and algorithms of COSY INFINITY.

The C++ interface is implemented through the Cosy class, which offers access from within C++ to the core of COSY INFINITY. This interfacing is achieved by embedding the COSY INFINITY execution engine into a C++ class. Since the glue that holds the two systems together is a very lightweight wrapper of C++ code, the performance of the resulting class is comparable with the performance of COSY INFINITY itself and exceeds that of other approaches (the CPU time lies within a factor of two to the regular COSY INFINITY system on most machines).

The COSY INFINITY language (c.f. §4) uses an object-oriented approach to programming which centers around the idea of dynamic typing: all Cosy objects have an internal type (which may be real, string, logical, etc. – refer to Appendix A for details) and the exact meaning of operations on Cosy objects is determined at runtime and not at compile time. Within a pure C++ framework, similar results could be achieved with a combination of inheritance and virtual functions.

The Cosy class attempts to be compatible with the C++ double precision data type. In most cases, it should be possible to convert an existing numerical program to a Cosy-based one by simply replacing the string “double” with the string “Cosy” in the source. However, using this approach would underutilize the Cosy class, which shows its real strengths if the advanced data types like intervals, DA vectors, or Taylor models are used. For example, replacing the double precision numbers in an existing program with Cosy objects that are initialized to DA vectors would allow high-order sensitivity analysis of the original program. Other benefits lie in the automatic verification of existing programs by using intervals or Taylor models.

7.1 Installation

The implementation of the Cosy class is based on the Fortran 77 files which make up the implementation of the COSY INFINITY system. Most of the actual C++ code is automatically generated from these Fortran 77 files by the F2C converter [13]. Consequently, use of the Cosy class requires the **F2C library** to be installed on the user's system. While the F2C library can be obtained from http://cm.bell-labs.com/netlib/f2c/, it is usually preferable to obtain a binary distribution of the library for a particular combination of compiler and system libraries (For the user's convenience, the source code of said library is also available from the COSY download section – however, this version may be not as up-to-date as the one available from other sources). It is important to note that the **F2C converter** is not required for the compilation of the Cosy class.

Several files of the distribution of the Cosy class are automatically generated from the Fortran 77 source files of COSY INFINITY by the F2C program. This conversion has been done by the COSY INFINITY development team and the users should never have to change any of the automatically generated files. Below is a description of the various automatically generated files contained in the distribution of the Cosy class.

***.cpp:** C++ source files automatically generated by F2C from the Fortran 77 source code

***.P:** include files automatically generated by F2C from the Fortran 77 files

***.c:** C-structs that are automatically generated from the Fortran 77 files by the F2C converter

The actual implementation of the Cosy class is contained in the files cosy.h and cosy.cpp. These files contain a small amount of specialized code (to interface with the automatically translated files mentioned above) and a large portion of these two files is automatically generated by the GENFOX program from the COSY INFINITY language description contained in the file GENFOX.DAT (c.f. Appendix A). The file main.cpp, which is part of the distribution, contains a small demo program that illustrates how the Cosy class can be used in practice. While it does not use all features of the class, it should provide a good starting point for the development of new programs with the Cosy class.

Finally, a Makefile is provided to compile the Cosy class and the file main.cpp to an executable "cosy". To start the compilation, just type "make cosy". The provided Makefile is rather generic and should be used as a starting point for a new build environment. If users port the build system to a new platform we would like to hear about this, so we can include the necessary files in the distribution. Currently, the Makefile is tailored to UNIX environments with the GNU make program **gmake** and GNU compiler.

7.2 Memory Management

The Cosy class manages its own internal memory and does not use dynamic allocation of memory by either malloc or new. To a large extent, this is the reason for the performance advantage that COSY INFINITY has over languages like C and C++.

As a consequence of this, every Cosy object requires a small portion of space in some non-dynamic memory region. While this is never an issue with global and local variables, this becomes an issue when Cosy objects are created dynamically by using `new` or `new[]`. Consequently, *dynamic allocation of Cosy objects should be avoided* whenever possible. If Cosy objects really have to be created dynamically, care should be taken to delete the objects as soon as possible, or the COSY system will exhaust its internal memory.

7.3 Public Interface of the Cosy Class

In this section we describe the public interface of the Cosy class. Most of the functions and operators described in this section fall in the categories of constructor, assignment, and unary operators and have no equivalent constructs in the standard COSY INFINITY language described in §4. Therefore, reading this section is essential for the understanding of the C++ interface to COSY INFINITY.

7.3.1 Constructors

To allow an easy conversion of existing code from the double data type to the Cosy data type, several constructors have been defined that should accommodate this through a variety of implicit constructions. Together with the built-in type conversions of C++, this mechanism should be able to handle almost any situation correctly.

```
Cosy( );
```

The default constructor creates a Cosy object with enough internal space to store one number or character. The object's type is initialized to **RE** and its value is set to zero.

```
Cosy( const double val, int len = 1 );
```

Create a Cosy object with enough internal space to hold `len` numbers or characters. The parameter `len` is optional and defaults to 1. The object's type is initialized to **RE** and its value is set to `val`.

```
Cosy( const int val, int len = 1 );
```

Create a Cosy object with enough internal space to store `len` numbers or characters. The parameter `len` is optional and defaults to 1. The type of the object is initialized to **RE** (COSY INFINITY does not have a dedicated data type for integers), and its value is set to `val`.

```
Cosy( const bool f );
```

Create a Cosy object with enough internal space to store one number or character. The object's type is initialized to **LO** and its value is set to the boolean value `f`.

`Cosy( const char *str );`

Create a Cosy object from a C string `str`. The object's type is set to **ST** and enough internal memory locations are allocated to hold the string (without the terminating NULL character, which is not needed in COSY). The object is initialized with the string `str`.

`Cosy( const Cosy& src );`

Create a new Cosy object from an existing one. The new object is initialized with a deep copy of `src`.

`Cosy( integer len, const int n, const int dim[] );`

This special constructor creates a Cosy object that represents a Cosy array of dimensionality `n`. The length of each of the dimensions is given in the array `dim`. And each entry of the array has internal space for `len` numbers and is initialized to zero with type **RE**. For further details on Cosy arrays, refer to section 7.6.

7.3.2 Assignment Operators

The Cosy class supports all assignment operations available in C++. Moreover, all the assignment operations that are commonly used with floating point numbers are implemented in a way compatible with the standard C++ definitions for floating point data types.

`Cosy& operator =(const Cosy& rhs)`

Assign a deep copy of `rhs` to the object and return a reference to it.

`Cosy& operator+=(const Cosy& rhs)`

Add `rhs` to the object and return a reference to it; equivalent to $x = x + rhs$.

`Cosy& operator-=(const Cosy& rhs)`

Subtract `rhs` from the object and return a reference to it; equivalent to $x = x - rhs$.

`Cosy& operator*=(const Cosy& rhs)`

Multiply the object with `rhs` and return a reference to it; equivalent to $x = x * rhs$.

`Cosy& operator/=(const Cosy& rhs)`

Divide the object by `rhs` and return a reference to it; equivalent to $x = x/rhs$.

`Cosy& operator&=(const Cosy& rhs)`

Unite the object with `rhs` and return a reference to it. For numerical Cosy objects, the result of a union is usually a vector. Please refer to Appendix A for further details. It should be noted that this implementation of this operator is not compatible with the default behavior of this operator in C++.

7.3.3 Unary Mathematical Operators

The Cosy class supports all unary operators available in C++. The operators are compatible with the default implementations for floating point variables.

`Cosy operator+()`

Return the positive of the object. This is in fact an identity operation and is included only for completeness.

`Cosy operator-()`

Return the negative of the object without modifying it.

`Cosy operator++()`

Add one to the object and return the result.

`Cosy operator--()`

Subtract one from the object and return the result.

`Cosy operator++(int)`

Add one to the object and return a copy of the object before the operation.

`Cosy operator--(int)`

Subtract one from the object and return a copy of the object before the operation.

7.3.4 Array Access

`Cosy get(const int coeff[], const int n)`

Obtain a copy of an array element. The element is described by the n-dimensional array coeff. More details on Cosy arrays are provided in §7.6.

`void set(const Cosy& arg, const int coeff[],const int n)`

Copy the Cosy object arg into an array. The target element is described by the n-dimensional array coeff. More details on Cosy arrays are provided in §7.6.

7.3.5 Printing, IO, and Streams

As indicated earlier, the code for the Cosy class is automatically derived from Fortran 77 code by using the F2C [13] converter. Consequently, the IO handling of the underlying C code is conceptually closer to the "printf"-type ideas of C than it is to the streams of C++.

However, by using temporary files, the Cosy class has partial support for the stream based IO of C++. This mechanism uses the file COSY.TMP in the current working directory as a translation buffer. This allows the Cosy class to be compatible with output streams.

```
friend ostream& operator<<(ostream& s, const Cosy& src)
```

Print a representation of the object src onto the ostream s. The printing uses the formats specified in §4.

7.3.6 Type Conversion

While the implicit type conversion mechanisms of C++ allow a transparent transition from the default C++ data types to Cosy objects. The conversion of Cosy objects into standard C++ data types on the other hand requires use of the dedicated conversion functions listed below.

```
friend double toDouble(const Cosy& arg)
```

Return a double precision variable that represents the result of calling the function **CONS** (c.f. §4) on the Cosy object arg.

```
friend bool   toBool  (const Cosy& arg)
```

Return a boolean variable that contains the boolean value of the Cosy object arg. If arg is not of type **LO**, the return value is undefined.

```
friend string toString(const Cosy& arg)
```

Return a C++ string object that contains the string contained in the Cosy object arg. If arg is not of type **ST**, the result is undefined.

7.4 Elementary Operations and Functions

The COSY INFINITY environment has a large number of operators and functions built into its language. The C++ interface to COSY INFINITY aims to give transparent access to these functions by trying to be compatible with both the notations of C++ and of COSY INFINITY. To that end, the operators are compatible with the C++ notations, and the elementary functions are compatible with the standard C++ naming conventions (and almost all functions defined in "math.h" for double precision floating point numbers are supported for Cosy objects).

As a general rule, all functions in C++ are named with the lower case version of their corresponding COSY INFINITY identifier. However, whenever COSY INFINITY uses a name for a function that does not exist in C++ (e.g., the absolute value function is called "abs" in COSY INFINITY, while it should be called "fabs" in C++), both names are made available. Whenever the name of a COSY FUNCTION clashes with reserved words of C++, the first letter of that function's name is capitalized (e.g. the COSY INFINITY function **REAL** is called "Real" in the C++ interface).

For the operators defined in COSY INFINITY, the following deviations from these general rules exist:

- While the exponentiation is an operation in COSY INFINITY, C++ uses the function `pow(...)` for this.
- The operator `#` of COSY INFINITY is not defined in C++ and has been replaced with the C++ operator `!=`.
- The operator `&` does not follow the standard C++ conventions and computes the union of two Cosy objects. However, since the Cosy class is meant to be used for the development of new programs, or as a replacement for double variables, overloading this operator is unlikely to cause any problems.

All operators and functions listed below have the following signature

```
inline <type> <name | operator op> (const Cosy& lhs, const Cosy& rhs);
```

Please refer to §A for further details on the individual functions and operators.

```
Cosy operator+
Cosy operator-
Cosy operator*
Cosy operator/
Cosy pow
bool operator<
bool operator>
bool operator==
bool operator!=
Cosy operator&


bool operator<=
bool operator>=
```

The standard functions defined for the Cosy class are listed below. These functions are also referred to as "intrinsic functions" for Cosy objects. To a large extent, the functions follow the standard naming conventions of standard C++. The first columns lists the COSY INFINITY name of the function and the second columns shows the complete C++ declaration of the function. For further details, the corresponding COSY INFINITY functions should be looked up in Appendix A.

TYPE	`Cosy type  ( const Cosy& x );`
LENGTH	`Cosy length( const Cosy& x );`
VARMEM	`Cosy varmem( const Cosy& x );`
VARPOI	`Cosy varpoi( const Cosy& x );`
NOT	`Cosy not   ( const Cosy& x );`
EXP	`Cosy exp   ( const Cosy& x );`
LOG	`Cosy log   ( const Cosy& x );`
SIN	`Cosy sin   ( const Cosy& x );`
COS	`Cosy cos   ( const Cosy& x );`
TAN	`Cosy tan   ( const Cosy& x );`
ASIN	`Cosy asin  ( const Cosy& x );`
ACOS	`Cosy acos  ( const Cosy& x );`
ATAN	`Cosy atan  ( const Cosy& x );`
SINH	`Cosy sinh  ( const Cosy& x );`
COSH	`Cosy cosh  ( const Cosy& x );`
TANH	`Cosy tanh  ( const Cosy& x );`
SQRT	`Cosy sqrt  ( const Cosy& x );`
ISRT	`Cosy isrt  ( const Cosy& x );`
SQR	`Cosy sqr   ( const Cosy& x );`
ERF	`Cosy erf   ( const Cosy& x );`
WERF	`Cosy werf  ( const Cosy& x );`
ABS	`Cosy fabs  ( const Cosy& x );`
ABS	`Cosy abs   ( const Cosy& x );`
NORM	`Cosy norm  ( const Cosy& x );`
CONS	`Cosy cons  ( const Cosy& x );`
RE	`Cosy re    ( const Cosy& x );`
IN	`Cosy in    ( const Cosy& x );`
LDB	`Cosy ldb   ( const Cosy& x );`
WIDTH	`Cosy width ( const Cosy& x );`
REAL	`Cosy Real  ( const Cosy& x );`
IMAG	`Cosy Imag  ( const Cosy& x );`
INT	`Cosy Int   ( const Cosy& x );`
NINT	`Cosy rint  ( const Cosy& x );`
NINT	`Cosy nint  ( const Cosy& x );`
DA	`Cosy da    ( const Cosy& x );`
CMPLX	`Cosy cmplx ( const Cosy& x );`
CONJ	`Cosy conj  ( const Cosy& x );`

7.5 COSY Procedures

The COSY INFINITY language environment has several procedures built into its language. These procedures range from diagnostic tools (e.g., **MEMFRE**) over file handling to complex tasks (e.g., **POLVAL**). For a complete interface from C++ to COSY INFINITY it was necessary to make these procedures available as "void functions". The C++ interfaces to the procedures all have a standardized signature:

```
void <name> (...);
```

All procedures take at least one argument, and all arguments are either of type "Cosy &" or "const Cosy &". The following table shows the names of the COSY INFINITY procedures in the first column, and the declaration of the corresponding C++ functions (a c parameter stands for "const Cosy &" arguments; v denotes "Cosy &" arguments).

MEMALL	`void memall ( v );`
MEMFRE	`void memfre ( v );`
OPENF	`void openf  ( c, c, c );`
OPENFB	`void openfb ( c, c, c );`
CLOSEF	`void closef ( c );`
REWF	`void rewf   ( c );`
BACKF	`void backf  ( c );`
READB	`void readb  ( c, v );`
WRITEB	`void writeb ( c, c );`
CPUSEC	`void cpusec ( v );`
QUIT	`void quit   ( c );`
SCRLEN	`void scrlen ( c );`
DAINI	`void daini  ( c, c, c, v );`
DANOT	`void danot  ( c );`
DAEPS	`void daeps  ( c );`
DAPEW	`void dapew  ( c, c, c, c );`
DAREA	`void darea  ( c, v, c );`
DAPRV	`void daprv  ( c, c, c, c, c );`
DAREV	`void darev  ( v, c, c, c, c );`
DAFLO	`void daflo  ( c, c, v, c );`
CDFLO	`void cdflo  ( c, c, v, c );`
RERAN	`void reran  ( v );`
DARAN	`void daran  ( v, c );`
DADIU	`void dadiu  ( c, v, v );`
DADER	`void dader  ( c, v, v );`
DAINT	`void daint  ( c, v, v );`
DAPLU	`void daplu  ( v, c, c, v );`

DAPEE	`void dapee  ( c, c, v );`
DAPEA	`void dapea  ( c, c, c, v );`
DAPEP	`void dapep  ( v, v, v, v );`
DANOW	`void danow  ( c, c, v );`
CDF2	`void cdf2   ( v, v, v, v, v );`
CDNF	`void cdnf   ( v, v, v, v, v, v, v, v );`
CDNFDA	`void cdnfda ( v, v, v, v, v, v, v );`
CDNFDS	`void cdnfds ( v, v, v, v, v, v, v );`
MTREE	`void mtree  ( v, v, v, v, v, v, v );`
LINV	`void linv   ( c, v, c, c, v );`
LDET	`void ldet   ( c, c, c, v );`
MBLOCK	`void mblock ( c, v, v, c, c );`
SUBSTR	`void substr ( c, c, c, v );`
STCRE	`void stcre  ( c, v );`
RECST	`void recst  ( c, c, v );`
VELSET	`void velset ( v, c, c );`
VELGET	`void velget ( c, c, v );`
VEZERO	`void vezero ( v, v, v );`
VELMAX	`void velmax ( c, v );`
VEFILL	`void vefill ( v, c, c, c, c );`
IMUNIT	`void imunit ( v );`
LTRUE	`void ltrue  ( v );`
LFALSE	`void lfalse ( v );`
INTERV	`void interv ( c, c, v );`
INSRND	`void insrnd ( c );`
INSRF	`void insrf  ( c );`
INLO	`void inlo   ( c, v );`
INUP	`void inup   ( c, v );`
IVVELO	`void ivvelo ( c, v );`
IVVEUP	`void ivveup ( c, v );`
IVSET	`void ivset  ( v, c, c );`
IVGET	`void ivget  ( c, c, v );`
INTPOL	`void intpol ( v, c );`
RDAVAR	`void rdavar ( c, c, c, c, c, c, v );`
RDVAR	`void rdvar  ( c, c, c, c, v );`
RDANOT	`void rdanot ( c, c, v );`
RDREA	`void rdrea  ( c, v );`
RDREAB	`void rdreab ( c, v );`
RDWRTB	`void rdwrtb ( c, c );`
DAEXT	`void daext  ( c, v );`
RDRBND	`void rdrbnd ( c, v );`
RDAREF	`void rdaref ( c, v );`
RDADOM	`void rdadom ( c, v );`
RDITIG	`void rditig ( c );`

RDNPNT	`void rdnpnt ( c );`
LDBP	`void ldbp   ( c, c, c, c, c, v, v, v );`
RDINT	`void rdint  ( c, c, v );`
RDPRIS	`void rdpris ( c );`
CLEAR	`void clear  ( v );`
GRMOVE	`void grmove ( c, c, c, v );`
GRDRAW	`void grdraw ( c, c, c, v );`
GRDOT	`void grdot  ( c, c, c, v );`
GRCURV	`void grcurv ( c, c, c, c, c, c, c, c, c, v );`
GRPROJ	`void grproj ( c, c, v );`
GRCHAR	`void grchar ( c, v );`
GRCOLR	`void grcolr ( c, v );`
GRWDTH	`void grwdth ( c, v );`
GRMIMA	`void grmima ( c, v, v, v, v, v, v );`
RKCO	`void rkco   ( v, v, v, v, v );`
POLVAL	`void polval ( c, c, c, c, c, v, c );`
POLSET	`void polset ( c );`
FOXDPR	`void foxdpr ( c, c );`
MEMWRT	`void memwrt ( c );`
VARMAX	`void varmax ( v );`

7.6 Cosy Arrays vs. Arrays of Cosy Objects

In the COSY INFINITY language environment, arrays are collections of objects that may or may not have the same internal type. Thus, within COSY INFINITY, it is conceivable to have an array with entries representing strings, intervals, and real numbers. In that sense, the notion of arrays in COSY INFINITY is quite similar to the notion of arrays of Cosy objects in C++.

However, there is a fundamental difference between the two concepts: a C++ array of Cosy objects is not a Cosy object. Due to this difference, the C++ interface does not use C++ arrays of Cosy objects (although the user obviously has the freedom to declare and use them). As a consequence, the interface provides two different (and slightly incompatible) notions of arrays. "Arrays of Cosy Objects" are C++ arrays and they can be used wherever C++ permits the use of arrays. "Cosy Arrays", on the other hand, are individual Cosy objects which themselves contain Cosy objects. Since several important procedures of COSY INFINITY assume their arguments to be Cosy arrays, Cosy arrays are quite important in the context of COSY INFINITY and its C++ interface.

Since the C++ interface to Cosy does not use the "[]" operator for the access to elements, users should use the utility functions

```
Cosy get(const int coeff[], const int n)
```

and

```
void set(const Cosy& arg, const int coeff[], const int n)
```

described in §7.3.4 to access the elements of a Cosy array. To simplify the access to individual array elements, we suggest that users use inheritance or external utility functions like

```
   Cosy get(Cosy &a, int i, int j) {
       int c[2] = {i+1, j+1};
       return a.get(c, 2);
   }
```

for convenient access to the elements of Cosy arrays. Since Cosy arrays start at one, (as opposed to C++ arrays that start at 0), these utility functions could also be used to mask this implementational detail from the user. However, since the user's requirements on the dimensionality of Cosy arrays vary widely, the distribution of the C++ interface does not provide any of these convenience functions.

Finally, we point out that the two different concepts of arrays lead to the possibility of having C++ arrays of Cosy arrays – although it would be quite challenging to maintain a clear distinction between the various indices needed to access the individual elements.

8 The Fortran 90 Interface

The Fortran 90 interface to COSY INFNITY gives Fortran 90 programmers easy access to the sophisticated data types of COSY INFINITY. The interface has been implemented in the form of a Fortran 90 module, which has recently been used as part of the GlobSol [14] system for constrained global optimization.

8.1 Installation

Installation of the Fortran 90 interface module to COSY INFINITY requires a Fortran 90 compiler that is backwards compatible with Fortran 77.

The distribution contains the four Fortran 77 files that make up the COSY INFINITY system (c.f. §1.3 for details on how to compile these files). However, some changes have been made to these files to enable use in the Fortran 90 module: the small program VERSION has been used to un-comment all the lines that contain the string *FACE in columns 1 to 5 (and to comment all the lines containing the string *NORM in columns 73 to 80).

The actual implementation of the module is contained in the files `cosy.f90` and `cosydef.f90` which contain all the necessary interfaces to use COSY INFINITY from Fortran 90.

The file `main.f90`, which is part of the distribution, contains a small demo program that illustrates how the COSY module can be used in practice. While it does not use all features of the module, it should provide a good starting point for the development of new programs with the COSY module. Compilation of the demo program is accomplished by compiling the individual Fortran files and linking them to the executable program.

Lastly, a makefile is provided that eases the compilation by allowing the user to type "make cosy". The makefile has bee used on UNIX systems with the Digital Fortran compiler "fort" and can easily be adopted to other platforms. If users port the build system to a new platform, we would like to hear about this, so we can include the necessary files in the distribution.

8.2 Special Utility Routines

The Fortran 90 interface to COSY INFINITY uses a small number of utility routines for low-level access to the internals. In this section we describe these routines in detail.

`SUBROUTINE COSY_INIT [<NTEMP>] [<NSCR>] [<MEMDBG>]`

Initialize the COSY system. This subroutine has to be called before any COSY objects are used.

NTEMP sets the size of the pool of temporary objects and defaults to 20. This pool of variables is used for the allocation of temporary COSY objects. Since Fortran 90 does not support automatic destruction of objects, it is necessary to allocate all temporary objects beforehand and never deallocate them during the execution of the program. The pool is organized as a circular list; and in the absence of automatic destruction of objects, if the number of actually used temporary variables ever exceeds NTEMP, memory corruption will occur. It is the responsibility of the user to set the size appropriately.

NSCR defaults to 5000 and sets the size of the variables in the pool. Additionally, the subroutine **SCRLEN** is called to set the size of COSY's internal temp variables. MEMDBG may be either 0 (no debug output) or 1 (print debug information on memory usage). It should never be necessary for users of the Fortran 90 module to set MEMDBG.

Neither the size of the pool, nor the size of the variables in the pool can be changed after this call. (Refer to §8.7 for more details on the pool of temporary objects.)

`SUBROUTINE COSY_CREATE <SELF> [<LEN>] [<VAL>] [<NDIMS>] [<DIMS>]`

Create a variable in the cosy core. All COSY objects have to be created before they can be used! This routine allocates space for the variable and registers it with the COSY system. SELF is the COSY variable to be created.

LEN is the desired size of the variable SELF (it determines how many DOUBLE PRECISION values can be stored in SELF) and defaults to 1. If VAL is given, the variable is initialized to it (VAL defaults to 0.D0). Independent of the parameters LEN and VAL, the type of the variable is set to **RE**.

This routine can also be used for the creation of COSY arrays (see also §8.8). If NDIMS and DIMS are specified, the variable SELF is initialized to be an NDIMS-dimensional COSY array with length DIMS(I) in the i-th direction. Each entry of the array has length LEN and is initialized to VAL with type **RE**.

`SUBROUTINE COSY_DESTROY <SELF>`

Destruct the COSY object SELF and free the associated memory. If SELF hasn't been initialized with COSY_CREATE, the results of this are undefined.

`SUBROUTINE COSY_ARRAYGET <SELF> <NDIMS> <IDXS>`

Return a copy of an element of the array SELF. NDIMS specifies the dimensionality of the array and IDXS is an array containing the index of the desired element (refer to §8.8 for further details on COSY arrays).

`SUBROUTINE COSY_ARRAYSET <SELF> <NDIMS> <IDXS> <ARG>`

Copy the COSY object ARG into an element of the NDIMS-dimensional array SELF. The target is specified by the NDIMS-dimensional array IDXS which contains the index of the target (refer to §8.8 for further details on COSY arrays).

`SUBROUTINE COSY_GETTEMP <SELF>`

Return the address of the next available temporary object from the circular pool (buffer) of such objects. While the value of the returned variable is undefined, the type is guaranteed to be **RE**. Refer to §8.7 for more details.

`SUBROUTINE COSY_DOUBLE <SELF>`

Extracts the DOUBLE PRECISION value from the variable SELF by calling the function COSY function **CONS**.

`SUBROUTINE COSY_LOGICAL <SELF>`

Extracts the logical value from the variable SELF. If the type of SELF is not **LO**, the result of the operation is undefined.

`SUBROUTINE COSY_WRITE <SELF> [<IUNIT>]`

Writes the COSY variable SELF to the unit IUNIT (which defaults to 6). This function uses the same algorithms employed by the COSY procedure **WRITE** (c.f. §4.4).

`SUBROUTINE COSY_TMP <ARG>`

Return a temporary COSY object initialized with the value ARG (which may be either of type DOUBLE PRECISION or INTEGER). The main purpose of this function is for the temporary conversion of parameters to COSY procedures. As an example, consider the following two equivalent code fragments. They illustrate that the use of the function COSY_TMP leads to simpler and less error prone code.

```
    TYPE(COSY) :: A,B,X
    CALL COSY_CREATE(A)
    CALL COSY_CREATE(B)
    CALL COSY_CREATE(X,2)
    A = 2
    B = 5
    CALL INTERV(A,B,X)
    CALL COSY_DESTROY(A)
    CALL COSY_DESTROY(B)

    TYPE(COSY) :: X
    CALL COSY_CREATE(X,2)
    CALL INTERV(COSY_TMP(2),COSY_TMP(5),X)
```

8.3 Operations

The Fortran 90 interface to COSY INFINITY offers all operators that the standard COSY system offers. For the convenience of of the user, additional support functions are provided that allow mixed operations between built-in data types and the COSY objects. The following tables list all the defined operations between COSY objects and built-in types. All operations involving COSY objects return COSY objects.

Addition +		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Subtraction -		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY
	COSY	COSY

Multiplication *		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Division /		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Power **		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Comparison `.LT.`		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Comparison `.GT.`		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Comparison `.EQ.`		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Comparison `.NE.`		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY

Concatenation `.UN.`		
COSY	COSY	COSY
DOUBLE PRECISION	COSY	COSY
COSY	DOUBLE PRECISION	COSY
INTEGER	COSY	COSY
COSY	INTEGER	COSY
DOUBLE PRECISION	DOUBLE PRECISION	COSY
DOUBLE PRECISION	INTEGER	COSY
INTEGER	DOUBLE PRECISION	COSY
INTEGER	INTEGER	COSY

8.4 Assignment

The Fortran 90 interface to COSY INFINITY provides several assignment operations that allow an easy transition between built-in data types and COSY objects. This section lists all the defined assignment operators involving COSY objects.

COSY LHS = COSY RHS

Copies the COSY object RHS to LHS. If LHS hasn't been created yet, it will be created automatically.

DOUBLE PRECISION LHS = COSY RHS

Converts the COSY object RHS to the DOUBLE PRECISION number LHS by calling the function COSY_DOUBLE.

LOGICAL LHS = COSY RHS

Converts the COSY object RHS to the LOGICAL variable LHS by calling the function COSY_LOGICAL.

COSY LHS = DOUBLE PRECISION RHS

Copies the DOUBLE PRECISION variable RHS to the COSY object LHS. If LHS hasn't been created yet, it will be created automatically. The type of LHS will be set to **RE**.

COSY LHS = LOGICAL RHS

Copies the LOGICAL variable RHS to the COSY object LHS. If LHS hasn't been created yet, it will be created automatically. The type of LHS will be set to **LO**.

COSY LHS = INTEGER RHS

Copies the INTEGER variable RHS to the COSY object LHS. If LHS hasn't been created yet, it will be created automatically. The type of LHS will be set to **RE**.

8.5 Functions

The Fortran 90 interface to COSY INFINITY supports most of the functions supported by the COSY language environment. The following list shows all the functions that are supported for COSY objects by the Fortran 90 interface to COSY INFINITY. The left column shows the Fortran 90 name of the function and the right column shows the name of the function in the COSY INFINITY environment. Refer to Appendix A for details on the COSY INFINITY functions.

TYPE	**TYPE**
LENGTH	**LENGTH**
VARMEM	**VARMEM**
VARPOI	**VARPOI**
NOT	**NOT**
EXP	**EXP**
LOG	**LOG**

SIN	**SIN**
COS	**COS**
TAN	**TAN**
ASIN	**ASIN**
ACOS	**ACOS**
ATAN	**ATAN**
SINH	**SINH**
COSH	**COSH**
TANH	**TANH**
SQRT	**SQRT**
ISRT	**ISRT**
SQR	**SQR**
ERF	**ERF**
WERF	**WERF**
ABS	**ABS**
NORM	**NORM**
CONS	**CONS**
RE	**RE**
IN	**IN**
LDB	**LDB**
WIDTH	**WIDTH**
REAL	**REAL**
IMAG	**IMAG**
INT	**INT**
NINT	**NINT**
DA	**DA**
CMPLX	**CMPLX**
CONJ	**CONJ**

8.6 Subroutines

All the standard procedures of the COSY INFINITY language environment are available as subroutines from the Fortran 90 interface to COSY. The names and parameter lists of the subroutines match the names and parameter lists of the normal COSY INFINITY procedures.

Automatic argument conversion is not available. That means that all arguments have to be either previously created COSY objects or temporary COSY objects obtained from calls to COSY_TMP.

8.7 Memory Management

The COSY Fortran 90 module is based on the standard core functions and algorithms of COSY INFINITY. As such, it uses the fixed size memory buffers of COSY INFINITY for storage of COSY objects. While this fact is mostly hidden from the user, understanding this concept helps in writing efficient code.

When a COSY object is created by using the routine COSY_CREATE, memory is allocate in the internal COSY memory. This memory is not freed until the routine COSY_DESTROY is called for this object. Moreover, since COSY's internal memory is stack based (and not garbage collected), memory occupied by one object will not be freed until all objects that have been created at a later time have also been destroyed.

Since Fortran 90 does not have automatic constructors and destructors, all objects have to be deleted manually. While this is generally acceptable for normal objects, this is impossible to guarantee for temporary objects. To allow temporary objects in the COSY module, a circular buffer of temp. objects is created when the COSY system is initialized with COSY_INIT.

As an example on how the pool of temporary objects should be used, consider the following fragment of code that implements a convenience interface to the COSY procedure **RERAN**. Internally, the function CRAN obtains one object from the pool for its return value. This avoids the obvious memory leak that would result if it was creating a new COSY object.

```
FUNCTION CRAN()
  USE COSY_MODULE
  IMPLICIT NONE
  TYPE(COSY) :: CRAN
  CALL COSY_GETTEMP(CRAN)
  CALL RERAN(CRAN)
END FUNCTION CRAN
```

However, it has to be stressed that the fixed size of the pool of temporaries bears a potential problem: there is no check in place for possible exhaustion of the pool. In other words, the pool has to be sized large enough to accommodate the maximum number of temp. objects at any given time during the execution of the program. Since this number is easily underestimated, especially for deeply nested expressions, the buffer should be sized rather generously.

8.8 COSY Arrays vs. Arrays of COSY objects

In the COSY INFINITY language environment, arrays are collections of objects that may or may not have the same internal type. Thus, within COSY INFINITY, it is conceivable to have an array with entries representing strings, intervals, and real numbers. In that sense, the notion of arrays in COSY INFINITY is quite similar to the notion of arrays of COSY objects in Fortran 90.

However, there is a fundamental difference between the two concepts: a Fortran 90 array of COSY objects is not again a COSY object. Due to this difference, the Fortran 90 module does not use Fortran arrays of COSY objects (although the user obviously has the freedom to declare and use them). As a consequence, the interface provides two different (and slightly incompatible) notions of arrays. "Arrays of COSY Objects" are Fortran 90 arrays and they can be used wherever Fortran permits the use of arrays. "COSY Arrays", on the other hand, are individual COSY objects which themselves contain COSY objects. Since several important procedures of COSY INFINITY assume their arguments to be COSY arrays, COSY arrays are quite important in the context of COSY INFINITY and its Fortran 90 interface modules.

To access the elements of COSY arrays, users should use the utility routines

`SUBROUTINE COSY_ARRAYGET <SELF> <NDIMS> <IDXS>`

and

`SUBROUTINE COSY_ARRAYSET <SELF> <NDIMS> <IDXS> <ARG>`

Finally, we point out that the two different concepts of arrays lead to the possibility of having Fortran 90 arrays of COSY arrays – although it would be quite challenging to maintain a clear distinction between the various indices needed to access the individual elements.

9 Acknowledgements

For very valuable help with an increasing number of parts of the program, we would like to thank Meng Zhao, Weishi Wan, Georg Hoffstätter, Ralf Degenhardt, Khodr Shamseddine, Nina Golubeva, Vladimir Balandin, Béla Erdélyi, Lars Diening, Michael Lindemann, Jens von Bergmann, and Ralf Tönjes who all at various times were at Michigan State University. We would also like to thank numerous COSY users for providing valuable feedback, many good suggestions, and streamlining the implementation on various machines.

We would like to thank Kurt Overley and Tom Mottershead for their optimizer QOPT, Jay Dittmann for help with the simplex optimizer, and Jorge More for providing the public domain optimizer LMDIF. We would like to thank Roger Servranckx for providing the MAD to COSY converter. We would like to thank Felix Marti for writing the GKS graphics drivers.

Financial support was appreciated from the Deutsche Forschungsgemeinschaft, the University of Gießen, the SSC Central Design Group, Lawrence Berkeley Laboratory, Michigan State University, the National Superconducting Cyclotron Laboratory, the Alfred P. Sloan Foundation, the National Science Foundation, and the U.S. Department of Energy.

References

[1] M. Berz. COSY INFINITY Version 8.1 - user's guide and reference manual. Technical Report MSUHEP-20704, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2002. see also http://cosy.pa.msu.edu.

[2] K. Makino and M. Berz. COSY INFINITY version 8. *Nuclear Instruments and Methods*, A427:338–343, 1999.

[3] M. Berz. *Modern Map Methods in Particle Beam Physics.* Academic Press, San Diego, 1999. Also available at http://bt.pa.msu.edu/pub.

[4] M. Berz. Forward algorithms for high orders and many variables. *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, SIAM, 1991.

[5] M. Berz. Automatic differentiation as non-Archimedean analysis. In *Computer Arithmetic and Enclosure Methods*, page 439, Amsterdam, 1992. Elsevier Science Publishers.

[6] M. Berz. Arbitrary order description of arbitrary particle optical systems. *Nuclear Instruments and Methods*, A298:426, 1990.

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

[8] M. Berz. Differential algebra precompiler version 3 reference manual. Technical Report MSUCL-755, Michigan State University, East Lansing, MI 48824, 1990.

[9] Christian Bischof, Alan Carle, George F. Corliss, Andreas Griewank, and Paul Hovland. ADIFOR: Generating derivative codes from Fortran programs. *Scientific Computing*, 1(1):11–29, 1992.

[10] K. Makino and M. Berz. Remainder differential algebras and their applications. In M. Berz, C. Bischof, G. Corliss, and A. Griewank, editors, *Computational Differentiation: Techniques, Applications, and Tools*, pages 63–74, Philadelphia, 1996. SIAM.

[11] K. Makino. *Rigorous Analysis of Nonlinear Motion in Particle Accelerators.* PhD thesis, Michigan State University, East Lansing, Michigan, USA, 1998. Also MSUCL-1093.

[12] M. Berz and J. Hoefkens. Verified high-order inversion of functional dependencies and superconvergent interval Newton methods. *Reliable Computing*, 7(5):379–398, 2001.

[13] S. I. Feldman, David M. Gay, Mark W. Maimone, and N. L. Schreyer. A Fortran-to-C converter. Technical report, AT&T Bell Laboratories, Murray Hill, NJ 07974, 1995.

[14] Frank Fritz, Paul Thalacker, George F. Corliss, and R. Baker Kearfott. Globsol user guide. Technical Report, Department of Mathematics, Statistics and Computer Science, Marquette University, Milwaukee, Wisc., 1998. `http://www.mscs.mu.edu/~globsol/User_Guide`.

A The Supported Types and Operations

The language of COSY INFINITY is object oriented, and it is very simple to create new data types and define operations on them. Details about the types and operations are described in the language description data file GENFOX.DAT which is read by the program GENFOX, which then updates the source code of the compiler.

The first entry in GENFOX.DAT 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 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.

Below follows a list of all object types as well as a list of all the operands available for various combinations of objects, the available intrinsic functions, and the available intrinsic procedures. These lists are automatically produced in LaTeX format by the program GENFOX with each compiler update.

This information is current as of 13-Oct-02.

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
IV	Interval Vector
GR	Graphics
CD	Complex Differential Algebra Vector
RD	Remainder-enhanced Differential Algebra Object (Taylor model)

Now follows a list of all operations available for various combinations of types. For each operation, a relative priority (Pr.) is given which determines the hierarchy of the operations in expressions if there are no parentheses.

Operation	Pr.	Type			Comment
		Left	Right	Result	
+	3	RE	RE	RE	
+	3	RE	DA	DA	
+	3	RE	VE	VE	
+	3	RE	CM	CM	
+	3	RE	IN	IN	
+	3	RE	IV	IV	
+	3	RE	CD	CD	
+	3	RE	RD	RD	
+	3	LO	LO	LO	logical OR
+	3	DA	RE	DA	
+	3	DA	DA	DA	
+	3	DA	CM	CD	
+	3	DA	CD	CD	
+	3	VE	RE	VE	
+	3	VE	VE	VE	
+	3	VE	IN	IV	
+	3	CM	RE	CM	
+	3	CM	DA	CD	
+	3	CM	CM	CM	
+	3	CM	CD	CD	
+	3	IN	RE	IN	
+	3	IN	VE	IV	
+	3	IN	IN	IN	
+	3	IN	IV	IV	
+	3	IN	RD	RD	
+	3	IV	RE	IV	
+	3	IV	IN	IV	
+	3	IV	IV	IV	
+	3	CD	RE	CD	
+	3	CD	DA	CD	
+	3	CD	CM	CD	
+	3	CD	CD	CD	
+	3	RD	RE	RD	
+	3	RD	IN	RD	
+	3	RD	RD	RD	

Operation	Pr.	Type			Comment
		Left	Right	Result	
–	3	RE	RE	RE	
–	3	RE	DA	DA	
–	3	RE	VE	VE	
–	3	RE	CM	CM	
–	3	RE	IN	IN	
–	3	RE	IV	IV	
–	3	RE	CD	CD	
–	3	RE	RD	RD	
–	3	DA	RE	DA	
–	3	DA	DA	DA	
–	3	DA	CM	CD	
–	3	DA	CD	CD	
–	3	VE	RE	VE	
–	3	VE	VE	VE	
–	3	CM	RE	CM	
–	3	CM	DA	CD	
–	3	CM	CM	CM	
–	3	CM	CD	CD	
–	3	IN	RE	IN	
–	3	IN	IN	IN	
–	3	IV	RE	IV	
–	3	IV	IV	IV	
–	3	CD	RE	CD	
–	3	CD	DA	CD	
–	3	CD	CM	CD	
–	3	CD	CD	CD	
–	3	RD	RE	RD	
–	3	RD	RD	RD	

Operation	Pr.	Type			Comment
		Left	Right	Result	
*	4	RE	RE	RE	
*	4	RE	DA	DA	
*	4	RE	VE	VE	
*	4	RE	CM	CM	
*	4	RE	IN	IN	
*	4	RE	IV	IV	
*	4	RE	CD	CD	
*	4	RE	RD	RD	
*	4	LO	LO	LO	logical AND
*	4	DA	RE	DA	
*	4	DA	DA	DA	
*	4	DA	CM	CD	
*	4	DA	CD	CD	
*	4	VE	RE	VE	
*	4	VE	VE	VE	
*	4	CM	RE	CM	
*	4	CM	DA	CD	
*	4	CM	CM	CM	
*	4	CM	CD	CD	
*	4	IN	RE	IN	
*	4	IN	IN	IN	
*	4	IV	RE	IV	
*	4	IV	IV	IV	
*	4	CD	RE	CD	
*	4	CD	DA	CD	
*	4	CD	CM	CD	
*	4	CD	CD	CD	
*	4	RD	RE	RD	
*	4	RD	RD	RD	

Operation	Pr.	Type			Comment
		Left	Right	Result	
/	4	RE	RE	RE	
/	4	RE	DA	DA	
/	4	RE	VE	VE	
/	4	RE	CM	CM	
/	4	RE	IN	IN	
/	4	RE	IV	IV	
/	4	RE	CD	CD	
/	4	RE	RD	RD	
/	4	DA	RE	DA	
/	4	DA	DA	DA	
/	4	DA	CM	CD	
/	4	DA	CD	CD	
/	4	VE	RE	VE	
/	4	VE	VE	VE	
/	4	CM	RE	CM	
/	4	CM	DA	CD	
/	4	CM	CM	CM	
/	4	CM	CD	CD	
/	4	IN	RE	IN	
/	4	IN	IN	IN	
/	4	IV	RE	IV	
/	4	IV	IV	IV	
/	4	CD	RE	CD	
/	4	CD	DA	CD	
/	4	CD	CM	CD	
/	4	CD	CD	CD	
/	4	RD	RE	RD	
/	4	RD	RD	RD	

Operation	Pr.	Type			Comment
		Left	Right	Result	
^	5	RE	RE	RE	
^	5	VE	RE	VE	

Operation	Pr.	Type			Comment
		Left	Right	Result	
<	2	RE	RE	LO	
<	2	ST	ST	LO	

Operation	Pr.	Type			Comment
		Left	Right	Result	
>	2	RE	RE	LO	
>	2	ST	ST	LO	

Operation	Pr.	Type			Comment
		Left	Right	Result	
=	2	RE	RE	LO	
=	2	ST	ST	LO	

Operation	Pr.	Type			Comment
		Left	Right	Result	
#	2	RE	RE	LO	
#	2	ST	ST	LO	

Operation	Pr.	Type			Comment
		Left	Right	Result	
&	2	RE	RE	VE	
&	2	RE	VE	VE	
&	2	ST	ST	ST	
&	2	VE	RE	VE	
&	2	VE	VE	VE	
&	2	IN	IN	IV	
&	2	IN	IV	IV	
&	2	IV	IN	IV	
&	2	IV	IV	IV	
&	2	GR	GR	GR	

Now follows a list of 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

Function	Argument Type	Type of Result
TYPE	RE	RE
TYPE	ST	RE
TYPE	LO	RE
TYPE	DA	RE
TYPE	VE	RE
TYPE	CM	RE
TYPE	IN	RE
TYPE	IV	RE
TYPE	GR	RE
TYPE	CD	RE
TYPE	RD	RE

- LENGTH returns the momentary length of a variable in 8 byte blocks

Function	Argument Type	Type of Result
LENGTH	RE	RE
LENGTH	ST	RE
LENGTH	LO	RE
LENGTH	DA	RE
LENGTH	VE	RE
LENGTH	CM	RE
LENGTH	IN	RE
LENGTH	IV	RE
LENGTH	GR	RE
LENGTH	CD	RE
LENGTH	RD	RE

- VARMEM returns the current memory address of an object

Function	Argument Type	Type of Result
VARMEM	RE	RE
VARMEM	ST	RE
VARMEM	LO	RE
VARMEM	DA	RE
VARMEM	VE	RE
VARMEM	CM	RE
VARMEM	IN	RE
VARMEM	IV	RE
VARMEM	GR	RE
VARMEM	CD	RE
VARMEM	RD	RE

- VARPOI returns the current pointer address of an object

Function	Argument Type	Type of Result
VARPOI	RE	RE
VARPOI	ST	RE
VARPOI	LO	RE
VARPOI	DA	RE
VARPOI	VE	RE
VARPOI	CM	RE
VARPOI	IN	RE
VARPOI	IV	RE
VARPOI	GR	RE
VARPOI	CD	RE
VARPOI	RD	RE

- NOT returns the negation of a logical

Function	Argument Type	Type of Result
NOT	LO	LO

- EXP computes the exponential function

Function	Argument Type	Type of Result
EXP	RE	RE
EXP	DA	DA
EXP	VE	VE
EXP	CM	CM
EXP	IN	IN
EXP	IV	IV
EXP	RD	RD

- LOG computes the natural logarithm

Function	Argument Type	Type of Result
LOG	RE	RE
LOG	DA	DA
LOG	VE	VE
LOG	CM	CM
LOG	IN	IN
LOG	IV	IV
LOG	RD	RD

- SIN computes the sine

Function	Argument Type	Type of Result
SIN	RE	RE
SIN	DA	DA
SIN	VE	VE
SIN	CM	CM
SIN	IN	IN
SIN	IV	IV
SIN	RD	RD

- COS computes the cosine

Function	Argument Type	Type of Result
COS	RE	RE
COS	DA	DA
COS	VE	VE
COS	CM	CM
COS	IN	IN
COS	IV	IV
COS	RD	RD

- TAN computes the tangent

Function	Argument Type	Type of Result
TAN	RE	RE
TAN	DA	DA
TAN	VE	VE
TAN	IN	IN
TAN	IV	IV
TAN	RD	RD

- ASIN computes the arc sine

Function	Argument Type	Type of Result
ASIN	RE	RE
ASIN	DA	DA
ASIN	VE	VE
ASIN	IN	IN
ASIN	IV	IV
ASIN	RD	RD

- ACOS computes the arc cosine

Function	Argument Type	Type of Result
ACOS	RE	RE
ACOS	DA	DA
ACOS	VE	VE
ACOS	IN	IN
ACOS	IV	IV
ACOS	RD	RD

- ATAN computes the arc tangent

Function	Argument Type	Type of Result
ATAN	RE	RE
ATAN	DA	DA
ATAN	VE	VE
ATAN	IN	IN
ATAN	IV	IV
ATAN	RD	RD

- SINH computes the hyperbolic sine

Function	Argument Type	Type of Result
SINH	RE	RE
SINH	DA	DA
SINH	VE	VE
SINH	CM	CM
SINH	IN	IN
SINH	IV	IV
SINH	RD	RD

- COSH computes the hyperbolic cosine

Function	Argument Type	Type of Result
COSH	RE	RE
COSH	DA	DA
COSH	VE	VE
COSH	CM	CM
COSH	IN	IN
COSH	IV	IV
COSH	RD	RD

- TANH computes the hyperbolic tangent

Function	Argument Type	Type of Result
TANH	RE	RE
TANH	DA	DA
TANH	VE	VE
TANH	IN	IN
TANH	IV	IV
TANH	RD	RD

- SQRT computes the square root

Function	Argument Type	Type of Result
SQRT	RE	RE
SQRT	DA	DA
SQRT	VE	VE
SQRT	CM	CM
SQRT	IN	IN
SQRT	IV	IV
SQRT	RD	RD

- ISRT computes the reciprocal of square root

Function	Argument Type	Type of Result
ISRT	RE	RE
ISRT	DA	DA
ISRT	VE	VE
ISRT	IN	IN
ISRT	IV	IV
ISRT	RD	RD

- SQR computes the square

Function	Argument Type	Type of Result
SQR	RE	RE
SQR	DA	DA
SQR	VE	VE
SQR	CM	CM
SQR	IN	IN
SQR	IV	IV
SQR	CD	CD
SQR	RD	RD

- ERF computes the real error function erf

Function	Argument Type	Type of Result
ERF	RE	RE
ERF	DA	DA

- WERF computes the complex error function w

Function	Argument Type	Type of Result
WERF	CM	CM
WERF	CD	CD

- ABS computes the absolute value

Function	Argument Type	Type of Result
ABS	RE	RE
ABS	DA	RE
ABS	VE	RE
ABS	CM	RE
ABS	CD	RE

- NORM computes the norm of a vector

Function	Argument Type	Type of Result
NORM	DA	RE
NORM	VE	VE
NORM	CD	RE

- CONS determines the constant part

Function	Argument Type	Type of Result
CONS	RE	RE
CONS	DA	RE
CONS	VE	RE
CONS	CM	CM
CONS	IN	RE
CONS	IV	VE
CONS	CD	CM
CONS	RD	RE

- RE determines the real number part

Function	Argument Type	Type of Result
RE	RE	RE
RE	DA	RE
RE	RD	RE

- IN computes the interval bound

Function	Argument Type	Type of Result
IN	RE	IN
IN	VE	IN
IN	IN	IN
IN	IV	IN
IN	RD	IN

- LDB computes the interval bound using LDB

Function	Argument Type	Type of Result
LDB	RD	IN

- WIDTH computes the width of an interval bound

Function	Argument Type	Type of Result
WIDTH	IN	RE
WIDTH	RD	RE

- REAL determines the real part

Function	Argument Type	Type of Result
REAL	RE	RE
REAL	DA	DA
REAL	CM	RE
REAL	CD	DA
REAL	RD	RD

- IMAG determines the imaginary part

Function	Argument Type	Type of Result
IMAG	RE	RE
IMAG	DA	DA
IMAG	CM	RE
IMAG	CD	DA
IMAG	RD	RD

- INT determines the integer part

Function	Argument Type	Type of Result
INT	RE	RE

- NINT determines the nearest integer

Function	Argument Type	Type of Result
NINT	RE	RE

- DA returns the i th elementary DA vector

Function	Argument Type	Type of Result
DA	RE	DA

- CMPLX converts to complex

Function	Argument Type	Type of Result
CMPLX	RE	CM
CMPLX	DA	CD
CMPLX	CM	CM
CMPLX	CD	CD

- CONJ conjugates a complex number

Function	Argument Type	Type of Result
CONJ	RE	RE
CONJ	DA	DA
CONJ	CM	CM
CONJ	CD	CD
CONJ	RD	RD

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 OPENF (3 arguments) opens a file. Parameters are unit number, filename (string), and status (string, same as in FORTRAN open).
- Procedure OPENFB (3 arguments) opens a binary file. Parameters are unit number, filename (string), and status (string, same as in FORTRAN open).
- Procedure CLOSEF (1 argument) closes a file. Parameter is unit number.
- Procedure REWF (1 argument) rewinds a file. Parameter is unit number.
- Procedure BACKF (1 argument) backspaces a file. Parameter is unit number.
- Procedure READB (2 arguments) reads an 8 byte real number in binary form. The arguments are the unit number and the variable name.
- Procedure WRITEB (2 arguments) writes an 8 byte real number in binary form. The arguments are the unit number and the variable name.
- Procedure CPUSEC (1 argument) returns the elapsed CPU time in seconds since last midnight on VMS; the elapsed system time on PC; and the elapsed CPU time in the process on UNIX.
- Procedure QUIT (1 argument) terminates execution; Argument = 1 triggers traceback.
- Procedure SCRLEN (1 argument) sets the amount of space scratch variables are allocated with.
- 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 (return).
- Procedure DANOT (1 argument) sets momentary truncation order for DA.

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

- Procedure DAPEW (4 arguments) prints the part of DA vector that has a certain order in a specified variable. Arguments are unit, DA vector, column number, and order.
- Procedure DAREA (3 arguments) reads a DA vector. Arguments are the unit number, the variable name and the number of independent variables.
- Procedure DAPRV (5 arguments) prints an array of DA vectors. Arguments are the array, the number of components, maximum and current main variable number, and the unit number.
- Procedure DAREV (5 arguments) reads an array of DA vectors. Arguments are the array, the number of components, maximum and current main variable number, and the unit number.
- Procedure DAFLO (4 arguments) computes the flow of x' = f(x) for time step 1. Arguments: the initial condition, array of right hand sides, result, and dimension of f.
- Procedure CDFLO (4 arguments) same as DAFLO but with complex arguments.

- Procedure RERAN (1 argument) returns a random number between -1 and 1.
- Procedure DARAN (2 arguments) fills DA vector with random entries. Arguments are DA vector and fill factor.
- Procedure DADIU (3 arguments) performs a division by a unit vector if possible. Arguments are the number of the unit vector and the two DA vectors.
- 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 DAPLU (4 arguments) replaces power of independent variable I by constant C. Arguments are the DA or CD vector, I, C, and the resulting DA or CD vector.
- Procedure DAPEE (3 arguments) returns a component of a DA or CD vector. Arguments are the DA or CD vector, the id for the coefficient in TRANSPORT notation, and the returning real or complex number.

- Procedure DAPEA (4 arguments) performs same with DAPEE, except the id is specified by an array with each element denoting the exponent. The third argument is the size of the array.

- Procedure DAPEP (4 arguments) returns a parameter dependent component of a DA or CD vector. Arguments are the DA or CD vector, the coefficient id, number of variables, and the resulting DA or CD vector.

- Procedure DANOW (3 arguments) computes the order weighted norm of the DA vector in the first argument. Second argument: weight, third argument: result.

- Procedure CDF2 (5 arguments) Lets $\exp(: f_2 :))$ act on first argument in Floquet variables. Other Arguments: 3 tunes (2π), result.

- Procedure CDNF (8 arguments) Lets $1/(1 - \exp(: f_2 :))$ act on first argument in Floquet variables. Other Arguments: 3 tunes (2π), array of resonances with dimensions, result.

- Procedure CDNFDA (7 arguments) Lets $C_j^{\pm}$ act on the first argument. Other Arguments: moduli, arguments, coordinate number, total number, epsilon, and result.

- Procedure CDNFDS (7 arguments) Lets $S_j^{\pm}$ act on the first argument. Other Arguments: moduli, arguments, spin argument, total number, epsilon, and result.

- Procedure MTREE (7 arguments) computes the tree representation of a DA array. Arguments: DA array, elements, coefficient array, 2 steering arrays, elements, length of tree.

- Procedure LINV (5 arguments) inverts a quadratic matrix. Arguments are the matrix, the inverse, the number of actual entries, the allocation dimension, and an error flag.

- Procedure LDET (4 arguments) computes the determinant of a matrix. Arguments are the matrix, the number of actual entries, the allocation dimension, and the determinant.

- Procedure MBLOCK (5 arguments) transforms a quadratic matrix to blocks on diagonal. Arguments are matrix, the transformation and its inverse, allocation and momentary dimension.

- Procedure SUBSTR (4 arguments) returns a substring. Arguments are string, first and last numbers identifying substring, and substring.

- Procedure STCRE (2 arguments) converts a string to a real. Argument are the string and the real.

- Procedure RECST (3 arguments) converts a real or a complex or an interval to a string using a FORTRAN format. Arguments are the real (or complex or interval), the format, and the string.
- Procedure VELSET (3 arguments) sets a component of a vector of reals. Arguments are the vector, the number of the component, and the real value for the component to be set.
- Procedure VELGET (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 VEZERO (3 arguments) used in repetitive tracking to prevent overflow due to lost particle.
- Procedure VELMAX (2 arguments) returns the maximum element of a vector.
- Procedure VEFILL (5 arguments) fills a vector array for scanning. Arguments: VE array, domain IN or IV, number of dimension, number of division, number of random points.
- Procedure IMUNIT (1 argument) returns the imaginary unit i.
- Procedure LTRUE (1 argument) returns the logical value true.
- Procedure LFALSE (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 INSRND (1 argument) enables and disables outward rounding of intervals with the parameters 1 and 0 respectively. By default the rounding is enabled. Extra caution has to be used for disabling the outward rounding, because it will void the validated enclosure computation.
- Procedure INSRF (1 argument) sets the factor f. $f\varepsilon$ is the outward rounding constant for interval intrinsic functions, where ε is the software determined machine error, and is the outward rounding constant for the other interval rounding including the binary operations. By default f is 10. The last specified f is kept independent of INSRND.
- 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 upper bound.

- Procedure IVVELO (2 arguments) returns the lower bounds of an interval vector as real vector.

- Procedure IVVEUP (2 arguments) returns the upper bounds of an interval vector as real vector.

- Procedure IVSET (3 arguments) sets a component of an interval vector. Arguments are the interval vector, the number of the component, and the interval for the component to be set.

- Procedure IVGET (3 arguments) returns a component of an interval vector. Arguments are the interval vector, the number of the component, and on return the interval of the component.

- Procedure INTPOL (2 arguments) determines coefficients of Polynomial satisfying $P(\pm 1) = \pm 1$, $P^{(i)}(\pm 1) = 0$, $i = 1, ..., n$. Arguments: coefficient array, n.

- Procedure RDAVAR (7 arguments) creates an RDA object. Arguments: a DA, NO, NV, reference point RE or VE, domain IN or IV, bounds for remainder and each order IN or IV, resulting RDA. A casual use of RDAVAR may void the validated enclosure computation. Consult us, if it is necessary to use this procedure.

- Procedure RDVAR (5 arguments) creates the i-th identity RDA. Arguments: i, reference point RE or VE, domain IN or IV, computation mode IMD, and the resulting i-th identity RDA. If the mode is 1, the domain is normalized to [-1,1] and the reference is set as 0 with the scaling. The validated enclosure computation is supported fully if the domain and the reference are normalized. This can be achieved either by setting IMD=1, or by preparing the domain and the reference as [-1,1] and 0.

- Procedure RDANOT (3 arguments) truncates an RDA to a lower order RDA. Arguments: RDA to be truncated, truncation order, resulting RDA.

- Procedure RDREA (2 arguments) reads an RDA. Arguments are the unit number and the variable name.

- Procedure RDREAB (2 arguments) reads an RDA in binary form. Arguments are the unit number and the variable name.

- Procedure RDWRTB (2 arguments) writes an RDA in binary form. Arguments are the unit number and the variable name.

- Procedure DAEXT (2 arguments) extracts the DA part from an RDA. Arguments are the RDA and the resulting extracted DA vector.

- Procedure RDRBND (2 arguments) extracts the remainder bound interval from an RDA. Arguments are the RDA variable and the resulting remainder bound interval.
- Procedure RDAREF (2 arguments) extracts the reference points from an RDA. Arguments are the RDA variable and the reference point (in RE) or points (in VE).
- Procedure RDADOM (2 arguments) extracts the domain intervals from an RDA. Arguments are the RDA variable and the domain interval (in IN) or intervals (in IV).
- Procedure RDITIG (1 argument) sets the algorithm number of RDA tightening. Some of currently available algorithms are for inner estimate, thus not validated. Refer to [11].
- Procedure RDNPNT (1 argument) sets the total number of points for scanning for RDA tightening with real rastering algorithms. Refer to [11].
- Procedure LDBP (8 arguments) bounds a polynomial over an interval box with LDB (Linear Dominated Bounding) algorithm. Arguments are a polynomial (DA), the degree and the dimensionality of the polynomial, the domain intervals (IN or IV) and the tolerance demand. The bound (IN) and the numbers of iterations for the lower bound and the upper bound are returned.
- Procedure RDINT (3 arguments) performs an integration of an RDA. Arguments are the number with respect to which to integrate, the RDA to be integrated and the resulting RDA.
- Procedure RDPRIS (1 argument) sets the RDA output option. If the parameter is 1, the order bounds are output as well. By default, the parameter is 0.
- Procedure CLEAR (1 argument) clears a graphics object.
- Procedure GRMOVE (4 arguments) produces a graphics object containing just one move. Arguments are the three coordinates and the graphics object.
- Procedure GRDRAW (4 arguments) produces a graphics object containing just one draw. Arguments are the three coordinates and the graphics object.
- Procedure GRDOT (4 arguments) produces a graphics object containing one move and a dot. Arguments are the three coordinates and the graphics object.
- Procedure GRCURV (10 arguments) produces a graphics object containing just one B-Spline. Arguments are the three final coordinates, the three components of the initial tangent vector, the three components of the final tangent vector and the graphics object.

- Procedure GRPROJ (3 arguments) sets 3D projection angles to a graphics object. Arguments are phi and theta in degrees and the graphics object.
- Procedure GRCHAR (2 arguments) produces a graphics object containing just one string. Arguments are the string and the graphics object.
- Procedure GRCOLR (2 arguments) produces a graphics object containing just one color change. Arguments are the new color id and the graphics object.
- Procedure GRWDTH (2 arguments) produces a graphics object containing just one width change. Arguments are the new width id and the graphics object.
- Procedure GRMIMA (7 arguments) finds the minimal and the maximal coordinates in a graphics object, arguments are the object and the minima and maxima for x, for y, and for z.
- Procedure RKCO (5 arguments) sets the coefficient arrays used in the COSY eighth order Runge Kutta integrator.
- Procedure POLVAL (7 arguments) performs POLVAL. See section 6.3 for details.
- Procedure POLSET (1 argument) sets the POLVAL algorithm number. 0: expanded, 1: Horner.
- Procedure FOXDPR (2 arguments) prints a dump of a variable. Arguments are the unit number and the variable name.
- Procedure MEMWRT (1 argument) writes memory to file : I, NBEG, NEND, NMAX, NTYP, CC, NC in first line and CC, NC in others. Argument is the unit number.
- Procedure VARMAX (1 argument) reports the maximum number of variables.

Index