## Contents

1 Before Using COSY INFINITY  
  1.1 How to Avoid Reading this Manual .................................. 5  
  1.2 What is COSY INFINITY ........................................... 5  
  1.3 User’s Agreement .................................................. 6  
  1.4 How to Obtain Help and to Give Feedback .......................... 7  
  1.5 How to Install the Code ........................................... 7  
      1.5.1 Microsoft Windows PC ...................................... 7  
      1.5.2 Mac OS X .................................................. 7  
      1.5.3 Using GNU Compilers ...................................... 8  
      1.5.4 Standard UNIX systems .................................... 9  
      1.5.5 Source files ............................................... 9  
  1.6 Memory Usage and Limitations ................................... 10  
  1.7 Syntax Changes .................................................. 10  
  1.8 Future Developments ............................................. 11  

2 COSY Types ..................................................................... 12  
  2.1 Reals, Complex, Strings, and Logicals ............................. 12  
  2.2 Intervals ..................................................................... 13  
  2.3 Vectors and Interval Vectors ....................................... 13  
  2.4 DA Vectors .................................................................. 14  
  2.5 Taylor Models (RDA Objects) ....................................... 15  
  2.6 The Intrinsic Procedure POLVAL .................................. 15  
  2.7 Verification of COSY ................................................ 16  

3 COSYScrip ................................................................. 18  
  3.1 COSYScrip Syntax Table ............................................. 18  
  3.2 A Simple Example for COSYScrip .................................. 18  
  3.3 General Aspects of COSYScrip ..................................... 19  
  3.4 Program Segments and Structuring ................................ 19  
  3.5 Flow Control Statements ......................................... 21  
  3.6 Input and Output ................................................... 22
### CONTENTS

3.7 Parallel Environments Based on MPI ........................................ 25
   3.7.1 Compiling Parallel COSY ........................................ 25
   3.7.2 Routines ................................................................ 25
   3.7.3 Running Parallel COSY ........................................ 26

3.8 Error Messages .............................................................. 26

4 The C++ Interface ............................................................ 28
   4.1 Installation ................................................................ 28
   4.2 Memory Management .................................................. 29
   4.3 Public Interface of the Cosy Class .................................. 29
       4.3.1 Constructors .................................................... 29
       4.3.2 Assignment Operators ......................................... 30
       4.3.3 Unary Mathematical Operators .............................. 31
       4.3.4 Array Access .................................................... 31
       4.3.5 Printing, IO, and Streams .................................... 31
       4.3.6 Type Conversion ................................................ 32
   4.4 Elementary Operations and Functions ............................ 32
   4.5 COSY Procedures ....................................................... 33
   4.6 Cosy Arrays vs. Arrays of Cosy Objects .......................... 34

5 The Fortran 90 Interface ....................................................... 35
   5.1 Installation ................................................................ 35
   5.2 Special Utility Routines .............................................. 35
   5.3 Operations .............................................................. 37
   5.4 Assignment ............................................................ 39
   5.5 Functions ............................................................. 40
   5.6 Subroutines ........................................................... 40
   5.7 Memory Management ................................................. 40
   5.8 COSY Arrays vs. Arrays of COSY objects ...................... 41

6 Optimization, Graphics, and the Graphical User Interface ............ 42
   6.1 Optimization .......................................................... 42
       6.1.1 Optimizers ...................................................... 42
CONTENTS

6.1.2 Adding an Optimizer ........................................ 43
6.2 Graphics .................................................................. 44
  6.2.1 Simple Pictures .................................................. 44
  6.2.2 Supported Graphics Drivers .................................. 44
  6.2.3 Adding Graphics Drivers ...................................... 45
  6.2.4 The COSY Graphics Meta File ............................... 46
6.3 Graphical User Interface ........................................... 46
  6.3.1 Basic GUIs ....................................................... 47
  6.3.2 Advanced GUIs .................................................. 48
  6.3.3 GUI Command Reference ................................. 48
  6.3.4 GUI Layout ................................................... 51
  6.3.5 Examples ....................................................... 51
  6.3.6 Java GUI ....................................................... 52

7 Acknowledgements ..................................................... 52

A The Supported Types and Operations .............................. 55
  A.1 Objects .......................................................... 55
  A.2 Operators ....................................................... 56
  A.3 Intrinsic Functions ............................................. 62
  A.4 Intrinsic Procedures ........................................... 75
1 Before Using COSY INFINITY

1.1 How to Avoid Reading this Manual

This manual attempts to be a sufficiently complete description of the features of COSY INFINITY; but as such much of it will be unnecessary for many users much of the time. The following is a road map that may allow navigation of the information most efficiently.

1. New users of COSY interested in its inner workings beyond tools using COSY, read the remainder of this chapter
2. New users of COSY 9.1, install it on the system of your choice. Follow instructions in section 1.5 on page 7
3. COSYSScript language users, note the brief syntax table in section 3.1 on page 18
4. COSYSScript language users in particular disciplines, glance at the respective demo files. Beam physics: demo.fox, Rigorous computing: tm.fox, Others: contact us.
5. C++ Users, refer to 4 on page 28
6. F90 Users, refer to 5 on page 35

1.2 What is COSY INFINITY

COSY INFINITY is a system for the use of various advanced concepts of modern scientific computing. COSY is extensively verified, and currently has more than 1000 registered users. The COSY system consists of the following parts.

1. A collection of advanced data types for various aspects of scientific computing. The data types include

   (a) DA (as well as the related CD) for differential algebraic computations [3], as well as high-order, multivariate automatic differentiation [1] [4] [2]
   (b) TM, the Taylor Model [13] [11] [12] data type. Allows rigorous verified computation under suppression of dependencies. Includes the rigorous treatment of a remainder bound over a given domain. Also supported are the interval (IN) and interval vector (IV) data types
   (c) VE, the high-performance vector data type. Provides performance advantages in environments supporting hyperthreading, multiple cores, or shared memory parallelism based on OpenMP, and even leads to gains on conventional systems because of favorable memory localization. Also supported are the conventional types Real (RE), String (ST), Logical (LO), as well as a Graphics data type (GR).

   The types are highly optimized for speed and performance, including support of sparsity. For details, refer to [1] [3]. Objects are stored in a built-in dynamic memory management system such that an entire object is always consecutive for efficient memory access.

2. The COSYSScript environment to use these data types with the following features:

   (a) Scripting language is compiled and executed on the fly and highly optimized for turnaround. No need for linking, very low interpretative overhead. Ideal for simulation, control, and algorithm prototyping.
(b) Object oriented with polymorphism (dynamic typing)
(c) Local and global optimization (non-verified) built in at the language level

3. A C++ Interface to utilize the types in user’s programs
4. A F90 Interface to utilize the types in user’s programs

The environment is extensively verified, and currently has more than 1000 registered users.

The COSY INFINITY system is being used for the following tasks.

1. High-order multivariate Automatic Differentiation of functions written C++, F90, and COSYS\textit{c}ript with support for sparsity and checkpointing
2. Solution of ODEs, single point and flow (dependence on initial conditions), as well as DAEs, based on COSY’s differential algebraic tools [3]
3. Arithmetic with Levi-Civita Numbers, allowing rigorous arithmetic including infinitely small and infinitely large numbers. Support for differentials, delta functions, etc.
4. The tm.fox package for rigorous and verified computation with often significantly reduced dependency.
5. COSY-GO, a rigorous global optimizer based on Taylor Models and Interval methods
6. COSY-VI [15] [6] [14], a rigorous verified integrator based on approximate differential algebraic flows and Taylor Models
7. The cosy.fox package for advanced particle beam dynamics simulations. Applications include high-order effects in storage rings, spectrographs, electron microscopes. Supports general arrangements of electromagnetic fields, including fringe fields, time dependent fields, and measured field data (on surface for stability). Support for normal form analysis, symplectic tracking, rigorous long-term stability estimates [7], and various other applications. For more details, refer to [3].

1.3 User’s Agreement

COSY INFINITY can be obtained from MSU under the following conditions.

Users are required 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 COSYS\textit{c}ript portions of the code are not to be modified without our consent. This does not include the addition of new optimizers and new graphics drivers as discussed in section 6.2.2; 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 COSYS\textit{c}ript 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. 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.
1.4 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 can be contacted at support@cosyinfinity.org.

1.5 How to Install the Code

All the system files, manuals, and installation packages of COSY INFINITY are currently distributed from the COSY web site

http://cosyinfinity.org

Installation packages for Microsoft Windows PC and for Mac OS X are available. The Microsoft Windows PC package is compiled by Intel Fortran Compiler 11.1 for Windows and linked with the GrWin graphics library. The Mac OS X package is compiled by Intel Fortran Compiler 11.1 for Mac OS X and linked with the AquaTerm graphics library. Installation under UNIX, Linux, or Cygwin is also described.

1.5.1 Microsoft Windows PC

An optimized executable program for Microsoft Windows PC produced by the Intel Fortran Compiler 9.1 and linked with the GrWin graphics library is available, and its usage is recommended compared to user compilation. The compiled version has special optimization for Intel dual core architecture and other modern Intel processors.

To install and run COSY in Windows, an Intel-based PC running Microsoft Windows is required. So far COSY has been successfully tested on Windows XP Professional, Windows Vista and Windows 7. With minor modifications it will also run on Windows 2000, Windows 98 and ME. If help is needed with those operating systems and it is not possible to upgrade to a more modern version, please contact us.

To install COSY for Windows, simply download and run the COSY INFINITY 9.1 installer WinCOSY9_1.exe Installer Package and follow the instructions provided by the installation program.

COSY 9.1 will be installed to the directory that you specify, and your start menu will be updated with the COSY INFINITY 9.1 folder, which contains a Read Me file, which contains a quick start guide, a link to the COSY Website, and a link to COSY GUI that allows you to navigate and choose a .fox file to run with the COSY Graphical User Interface.

The installer will also associate all files with the .fox extension on your system to be run with the COSY GUI interface simply on a double click.

In addition, the environment variable COSY9.1PATH is set appropriately in the situation that you need to find COSY INFINITY 9.1 from the command line.

1.5.2 Mac OS X

An executable program for Mac OS X 10.5 and higher produced by the Intel Fortran Compiler 11.1 linked with the AquaTerm graphics library is available, and its usage is recommended compared to user compi-
The compiled version supports all Intel-based Macintosh hardware and has special optimization for Intel dual core architecture and other modern Intel processors.

To install and run COSY on Mac OS X, an Intel-based Apple Mac running Mac OS X 10.5 or higher is required. So far COSY has been successfully tested on Mac OS X 10.6.5. It is not possible to run COSY on older PowerPC based systems. Support for earlier versions of Mac OS X is possible with some modifications.

To use COSY on Mac OS X, it is necessary to install AquaTerm, a freely available graphical terminal used by many plotting tools such as Gnuplot or PGPLOT on Mac OS X. PLEASE NOTE: As of December 2010 the version that can be downloaded from their web site is not fully compatible with both 32 and 64 bit programs, so we provide a slightly modified version included in the COSY installation package instead.

It is recommended to also install the Apple Developer Tools, to obtain access to their editor known as Xcode. The Developer Tools are included on a CD, but they can also be downloaded for free from Apple’s Developer Web site. COSY for Mac comes with a little extension for Xcode that enables syntax highlighting for editing COSY files with Xcode.

To install COSY for Macintosh, simply download the MacCOSY9.1.zip Installer Package and follow the instructions provided by the installation program. This will install COSY, AquaTerm and the FOXY language plug-in for Xcode.

A folder called COSY INFINITY 9.1 will be created in your Applications folder. Within that folder there is a Read Me file containing a quick start guide, and a program called "Run COSY" to start COSY. This program will automatically be associated with .fox files so to start COSY you can simple right-click any fox file and select "Run COSY" from the "Open With" menu.

The COSY installer also sets up a link to the cosy command line executable in your system. You can use it by starting the Terminal program in the Utilities folder in Applications and typing "cosy". This method is only recommended for advanced users.

1.5.3 Using GNU Compilers

The following is an example “Makefile” to compile and link the program with the PGPLOT graphics library using the GNU Fortran 77 compiler g77.

```
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)
```

It is advised to check the documentation of the GNU Fortran compiler about platform specific options. The compiler optimization options are not recommended for GNU Fortran compilers, because it sometimes causes trouble in handling the COSY syntax. For the use on Intel-based PCs, it is to be noted that the use of the GNU compilers carries performance penalties around a factor of three compared to the use of code generated with the Intel compiler mentioned in the sections on Windows and Macintosh installation. If desired, the GrWin graphics library also can be linked.
1.5.4 Standard UNIX systems

The Fortran source is by default compatible with standard UNIX systems. This also includes the standard Cygwin environment. In general, the default compiler optimization is recommended.

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

1.5.5 Source files

If it is necessary to install COSY INFINITY in a GNU or UNIX environment, or on any other platform, the source files are available at the above web site.

The code for COSY INFINITY consists of the macro source files, which depend on the particular application of COSY. For use in Beam Physics, the code cosy.fox is needed, for applications to rigorous computing, other files are needed. The respective macro files are written in COSY’s own language and have to be compiled by the local executable program of COSY INFINITY as part of the installation process.

Fortran Source Files

- foxy.f
- dafox.f
- foxfit.f
- foxgraf.f

The four files foxy.f, dafox.f, foxfit.f and foxgraf.f are written in standard Fortran 77 and have to be compiled and linked. foxy.f is the compiler and executor of COSYScript. dafox.f contains the routines to perform operations with objects, in particular the differential algebraic routines, Taylor Model routines, and the interval package. foxfit.f contains the package of nonlinear optimizers. foxgraf.f contains the available graphics output drivers, which are listed in section 6.2.2.

The distributed source file foxgraf.f assumes to be linked with a local PGPLOT library. If a local PGPLOT library is not available, the source file foxgraf.f has to be modified. If a local GrWin library is available, the library can be linked after modifying foxgraf.f. See section 6.2.2 for the details.

All the Fortran parts of COSY INFINITY are written in standard ANSI Fortran 77. However, certain aspects of Fortran 77 are still platform dependent; in particular, this concerns file handling, command line handling, and the CPU time measurement. All system dependent features of COSY INFINITY are coded for various operating systems, including UNIX, Linux, Microsoft Windows PC, Mac OS X, Opteron, IBM.

As of December 2010, all commonly available Fortran compilers have identical file handling. The following compilers have been verified for compatibility with the COSY system and used for production runs.

- Intel Fortran Compiler versions 9.2 and 11.1 for Microsoft Windows, Linux, and Mac OS X
- Compaq/DEC Fortran Compiler for Microsoft Windows
- GNU Fortran Compiler for Linux, and for Cygwin under Microsoft Windows
1.6 Memory Usage and Limitations

COSY is written in such a way that with modern compilers, including those used for the downloadable Windows and Mac OS X versions, memory is allocated dynamically as needed up to a certain maximum. At start-up, COSY requires approximately 30MB of physical memory; with conventional loads in typical beam physics and rigorous computing environments, the actually used memory rarely exceeds 50MB. However, as currently configured, COSY will allocate approximately 1.5GB of memory at startup. If that is a problem, it is perfectly acceptable to have most of this memory supplied by a swap file on disk as it is rarely needed. Should this be not enough for certain really large applications, the memory that can be allocated can be increased up to the limit of the utilized environment by changing the parameter LMEM in all occurrences in foxy.f, dafox.f and foxgraf.f to a higher value.

1.7 Syntax Changes

With very minor exceptions, version 9.1 is downward compatible to the previously released versions of COSY INFINITY, and any user deck for version 6 and higher should run under versions 9.1.

Changes in version 9.0

The naming of Taylor Model related routines are changed from “RD” to “TM”, and the syntax of some Taylor Model routines have been changed slightly. Refer to Appendix A and pages 15 and 81.

The COSY convenience functions for intervals INL and INU are now COSY intrinsic functions. So, there is no need to define functions INL and INU in any fox files.

Several old graphics drivers in foxgraf.f have been removed. The removed drivers are GKS, VGA, graPHIGS and Tektronix.

Changes in version 9.1

The COSY GUI environment has been added to facilitate the development of professional graphical user interfaces from within COSY SCRIPT. See Section 6.3.

The maximum line length of COSY SCRIPT files has been expanded from 80 characters to 4096 characters. Furthermore, the tab character has been added as a valid white space character in the COSY syntax.

COSY INFINITY does not generate .cod files for each executed .fox file any more. Code listings in .lis files are still generated as before.

When reading input from a file, GUI, or the keyboard trailing spaces are now automatically removed from the input if it is returned as a string. We believe this change should not affect existing code.
1.8 Future Developments

A variety of additional features are currently under development and/or alpha testing and are expected to become available in a future version. Even before the official release, they may be available for use by collaborators. Some of the features under development are

1. Arbitrary precision and rigorous data types and operations for DA, Taylor models, and intervals
2. Enhanced non-verified optimization tools, primarily genetic algorithms
3. Direct language-level interface to the rigorous verified global optimizers COSY-GO
4. Direct language-level interface to a new hybrid differential algebraic ODE integrator as a further development of COSY-VI
5. Fully rigorous tools for theorem proving in Dynamical Systems, including enclosures for attractors, stable and unstable manifolds, homoclinic and heteroclinic points, Poincare sections, normal forms, automatic bounds for topological entropy, and others.
2 COSY 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, also called polymorphism. Thus, the same expression can be evaluated with different types, and the same variable can assume different types at different times in the execution.

In this section, we will discuss the corresponding COSY functions and procedures that allow the explicit initialization of COSY variables to various types, and illustrate some of the most important tools for the manipulation of these types.

All examples are given in COSYScript, but readily translate to the syntax of C++ and/or F90, using the same names for intrinsic functions and procedures.

2.1 Reals, Complex, Strings, and Logicals

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

\begin{verbatim}
VARIABLE X 1 ; VARIABLE Y 1 ;
X := 1 ; \{Assigns value 1 to variable X\}
Y := EXP(-3) ;
\end{verbatim}

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

Complex numbers are created with the help of the COSY intrinsic function \texttt{CM}. The following two fragments each create a variable \texttt{Z} and initialize it to \(z = 2 - 3i\). Note that the variables \texttt{Z} and \texttt{I} have to be declared with enough space to hold two double precision numbers.

\begin{verbatim}
VARIABLE Z 2 ; VARIABLE I 2 ; VARIABLE X 1 ; VARIABLE Y 1 ;
I := CM(0&1) ; \{Assigns imaginary unit to variable \texttt{i}\}
Z := 2 - 3*I ; \{Assigns complex result by mixing real and complex\}
\end{verbatim}

or

\begin{verbatim}
Z := CM(2&(-3)) ; \{Assigns complex number (2,-3) directly\}
X := RE(Z) ; \{Determines the real part of \texttt{Z}\}
X := Z|1 ; Y := Z|2 \{Extracts the real and imaginary parts of \texttt{Z}\}
\end{verbatim}

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

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

VARIABLE S 80 ; VARIABLE T 80 2 ;
T(1) := 'HELLO ' ; {Assigns values to strings}
T(2) := 'WORLD' ;
S := T(1)&T(2) ; {Concatenates the two strings}
S := ST(4*atan(1)) ; {Contains an approximation of the leading digits of PI}

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.

Logical variables can be created by assignment using operators that return results of type logical, or by the use of the intrinsic function $LO$ described in Appendix A. The following code fragments illustrates this:

VARIABLE L 1 ;
L := 1=1 ;
L := LO(1) ;

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

2.2 Intervals

Interval variables can be created by using the COSY procedure intrinsic function $IN$. The following fragment creates a variable $I$ and initializes 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 ;
I := IN((-2)&3) ; {Creates interval $[-2,3]$}

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 verified enclosure computation.

The following are some frequently used intrinsic functions about intervals:

$X := INL(I)$ ; {Returns lower bound as a Real}
$X := INU(I)$ ; {Returns upper bound as a Real}
$X := I|1$ ; {Returns lower bound as Real}
$X := I|2$ ; {Returns upper bound as Real}
$X := RE(I)$ ; {Returns midpoint as a Real}

2.3 Vectors and Interval 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).
The appropriate use of vectors allow performance gains on processors utilizing hyperthreading or multiple cores, in OpenMP environments, and also in other environments due to simplifications in memory access.

Several different vector types exist, distinguished by the type of the components. Vectors can be 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:

```plaintext
VARIABLE V 4 ; VARIABLE X 1 ;
V := 22&33 ; {Creates Vector V from two components 22 and 33}
V := 11&V&44 ; {Turns V into a vector with four components}
X := V|3 ; {Extracts third component from V and stores in X}
X := VMIN(V) ; {Returns the minimum of the entries in V}
X := VMAX(V) ; {Returns the maximum of the entries in V}
```

```plaintext
X := RE(V) ; {Computes the arithmetic mean of the entries of V}
I := IN(V) ; {Generates Interval from the range of components}
```

```plaintext
VARIABLE II 8 ; VARIABLE I 2 ;
I := IN(2&3) ; {Generates Interval (from vector of length two)}
II := I&(2*I) ; {Generates Interval Vector with two components}
II := II&(II/3) ; {Generates Interval Vector with four components}
II := II - (II|1&II|2&II|3&II|4) ; {Disassembles II into components, reassembles them, subtracts from II}
```

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

### 2.4 DA Vectors

DA vectors can be created in several ways. First, it is important to distinguish DA Vectors from the usual vector data types: DA vectors are multiplied according to the rule of an algebra (in fact, a differential algebra), while Vectors are multiplied component-wise. Also, DA vectors support the derivation and anti-derivation operations characteristic of differential algebraic structures.

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. 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.

```plaintext
VARIABLE D 100 ; VARIABLE NM 1 ;
DAINI 3 2 0 NM ; {Initializes DA for order 3 and 2 variables}
D := DA(1)*EXP(DA(1)+DA(2)) ; {Assigns D to be a DA vector}
```

The differential algebraic structure induces a derivation and an anti-derivation operation. These can be used in the following way:

```plaintext
VARIABLE D2 100 ; VARIABLE DI 100 ;
```
2.5 Taylor Models (RDA Objects)

D2 := D%2 ; {Assigns D2 to be the DA vector of the partial
derivative of D with respect to variable 2}

DI := D%(-1) ; {Assigns DI to be the DA vector of the integral
of D with respect to variable 1}

It is possible to extract individual coefficients from DA vectors:

X := RE(D2) ; {Extracts constant part from D2}
X := DI|(2&1) ; {Extracts coefficient x^2 y from DI}

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

2.5 Taylor Models (RDA Objects)

Taylor Model

Taylor Model variables [11] [13] [12] should be created evaluating expressions with elementary Taylor Models. The latter can be created with the intrinsic procedure TMVAR (Refer to page 81) or the convenience function TMI. Like in the case of DA vectors, use of Taylor Models requires prior initialization of the DA system. The following fragment creates a 10th order Taylor Model for \( f(x_1, x_2) = x_1 \cdot \exp(x_1 + x_2) \), defined over the domain \( (2 + [-1/4, 1/4]) \times (5 + [-1/2, 1/2]) \) with reference point of \( (2, 5) \)

VARIABLE D 1000 ; VARIABLE NM 1 ;
VARIABLE X1 100 ; VARIABLE X2 100 ;

DAINI 10 2 0 NM ;
X1 := 2 + TM(1)/4 ; X2 := 5 + TM(2)/2 ;
D := X1*EXP(X1+X2) ;

Coefficients from Taylor models can be extracted in the same way as for DA vectors. Bounds for the orders and remainders can be obtained with the operation

IVB := IV(D) ; {Loads IVB as an interval vector with 12 intervals,
where the first n intervals are interval bounds
for the polynomial parts of orders 0 through n,
and the (n+1)st interval is the remainder interval}
I := IN(D) ; {Computes an interval bound for D}
I := LDB(D) : {Computes a bound of D by LDB. Usually significantly
sharper than the bound obtained by IN}

More details on the operations and functions defined for Taylor Models are given in Appendix A.

2.6 The Intrinsic Procedure POLVAL

An important COSY intrinsic procedure for DA vectors and Taylor Models is the tool POLVAL. It has the formal syntax

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 has already been called with L= 1, and if it is called with the same polynomial array P again, a certain part of internal analysis of P can be avoided by calling POLVAL with L= −1 or L= 0. (There are other advanced settings for L, but their use is discouraged for normal users because they may interfere with the internal use of POLVAL of various COSY tools.)

The type of the array A is free, but all 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 an efficient vectorizing polynomial evaluator, which is used for example for repetitive tracking in beam physics applications. If necessary, adding 0*A(1) to subsequent array elements A(I) can make the type of the argument array element agree to that type of A(1).

In the case the Array P is of Taylor Model type, checks are made to ensure that the ranges of the arguments are contained in the domains of the TMs in P. Moreover, the results are represented in one of the verified data types (intervals, interval vectors, Taylor Models) and the result is rigorous.

2.7 Verification of COSY

The operations on the various types have been verified for correctness in a variety of ways.

- The intrinsic operations of the Real, Complex, and DA data types have been verified for various complex examples in Beam Physics against the code COSY 5.0 [5]. Despite the similar name, COSY 5.0 uses analytic formulas developed by a custom-made high performance formula manipulator [8] and not DA tools to compute flows of particle accelerators up to order five. Agreement to near machine precision has been obtained for all terms in the flow expansion up to order five for a large class of different particle optical systems. Since the computation of these flow expansions requires virtually all COSY intrinsic operations and functions for the Real, Complex, and DA data types, any errors in their implementation would be expected to lead to some discrepancies. Since all operations in the DA data types are independent of order, agreement of up to order five also provides confidence for agreement to higher order.

- Flows for various specific ODEs that possess certain invariants of motion have been cross checked against these invariants. In particular, a large class of flows of systems in Beam Physics up to orders 15 has been checked for satisfaction of symplecticity as well as energy conservation. Similar to the previous test, any errors in implementation of the Real, Complex, and DA data types would be expected to lead to violations of these invariants.

- Advanced arguments involving symplectic representations and geometric symmetries allow to devise nonlinear systems for which all nonlinearities of the flows up to a given order cancel at certain values of the independent variable [17] [18]. Following these prescriptions, such systems have been designed with COSY, and as predicted in the theory, the advertised nonlinearities do indeed vanish [19]. This provides confidence in the ability to compute the underlying flows properly, and again provide confidence in their correctness.

- The interval verified data types have been compared in rather extensive tests against high-precision arithmetic packages by Corliss and Yun [9]. The Taylor model data types have been verified in the same manner. Further extensive automated tests have been performed by Nathalie Revol against other high-precision packages (unpublished). The theoretical soundness of their implementation has
been verified [16]. Since the underlying Taylor models utilize those of the DA type, this also provides verification of those operations.
3 COSYScript

The COSYScript language is based on a minimal and compact syntax. Experience shows that the COSY Syntax Table combined with some examples usually allow users to work with COSYScript within minutes.

COSYScript is object oriented with parametric polymorphism (dynamical type assignment). The language is compiled and linked to a meta-format on the fly and immediately executed. Combined with the ability to include pre-compiled code, this leads to a very rapid turnaround from input completion to execution. Combined with built-in tools for optimization, this makes the tool particularly suitable for simulation, as a control language, and for fast prototyping.

Great emphasis is put on performance, evidenced by negligible overhead to the cost of the operations on the types. COSYScript usually outperforms code based on the C++ and F90 interfaces discussed in further sections.

3.1 COSYScript Syntax Table

BEGIN ; END ;
VARIABLE <name> <length> ;
PROCEDURE <arguments> ; ENDPROCEDURE ;
FUNCTION <arguments> ; ENDFUNCTION ;

<name> := <expression> ; (Assignment)

IF <expression> ; ELSEIF <expression> ; ENDIF ;
WHILE <expression> ; ENDWHILE ;
LOOP <name> <beg> <end> ; ENDLOOP ;
PLOOP <name> <beg> <end> ; ENDPLOOP <comm. rules> ;
FIT <variables> ; ENDFIT <parameters, objectives> ;

WRITE <unit> <expressions> ; READ <unit> <names> ;
SAVE <filename> ; INCLUDE <filename> ;

3.2 A Simple Example for COSYScript

BEGIN ; VARIABLE NM 1 ; VARIABLE W 1 ; VARIABLE I 1 ;
FUNCTION FUN X Y ; FUN := SQR(SIN(X-2*Y)) + SQR(COS(X-2*Y)) ; ENDFUNCTION ;
DAINI 10 2 0 NM ; WRITE 6 'Maximum number of Coefficients: '&ST(NM) ' ;
WRITE 6 'Real Evaluation: '&ST(FUN(2,3)) ;
WRITE 6 'Complex Evaluation: '&ST(FUN(CM(1&2),CM(3&1))) ;
WRITE 6 'Vector Evaluation: ' FUN(2&4&6 ,3&5&7) '' ;
WRITE 6 'DA Vector Evaluation: ' FUN(2+0.1*DA(1),3+0.1*DA(2)) ;
LOOP I 1 4 ; W := 2^(-I) ; WRITE 6 ' *** W = '&ST(W) ;
WRITE 6 'Interval Evaluation: ' FUN(2+W*IN((-1)&1),3+W*IN((-1)&1)) ;
WRITE 6 'Taylor Model Evaluation: ' FUN(2+W*TM(1),3+W*TM(2)) ;
ENDLOOP ; END ;
3.3 General Aspects of COSYScript

Most commands of COSYScript consist of a keyword, followed by expressions and names of variables, and terminated by a semicolon. The individual entries 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 placed 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.

3.4 Program Segments and Structuring

COSYScript 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 files and ends at their 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 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. Various convenience functions to determine these for the COSY types are available; but if the information is provided directly, a real or double precision number requires a length of 1, a complex double precision number or an interval a length of 2. A DA vector requires a length of at least the number of partial derivatives \((n + v)!/(n! \cdot v!))\) in \(v\) variables to order \(n\) to be stored, a CD vector requires twice that, and a TM requires that plus \(2n + 2v\). Note that during allocation, the type is initialized to Real, and the value set to 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, and in this manner it is possible to emulate user-defined objects. As an example, the command

\[
\text{VARIABLE X 100 5 7 ;}
\]

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

\[
<\text{variable or array element}> := <\text{expression}> ;
\]

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

\[
<\text{procedure name}> \{ <\text{expression}> \} ;
\]

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

\[
<\text{function name}> ( <\text{expression}> \{ <, \text{expression}> \} ) ;
\]

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.

3.5 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 Logical 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 Logical 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 COSYScript, but the same effect can be achieved with the statement ELSEIF LO(1) ; where LO is a convenience function that returns True and False for arguments 1 and 0, respectively.

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 COSYScript 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 < ϵ > < Nmax > < Nalgorithm > {< Objective(s)> } ;

Here the names denote the visible variables that are being adjusted. ϵ is the tolerance to which the minimum is requested. Nmax 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. Nalgorithm gives the number of the optimizing algorithm that is being used. For the various optimizing algorithms, see section 6.1 (page 42). < Objective(s)> 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 of the program 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.

3.6 Input and Output

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

READ <expression> <name> ;
and

\texttt{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. Special unit numbers are provided for input and output to the Graphical User Interface (see Section 6.3). Unit numbers can be associated with particular file names by using the \texttt{OPENF} and \texttt{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 \texttt{READ} and \texttt{WRITE}. Use \texttt{READB} and \texttt{WRITEB} instead.

\texttt{READB <expression> <name> ;}

\texttt{WRITEB <expression> \{<expression>\} ;}

The Taylor Model binary input and output use the procedures \texttt{RDREAB} and \texttt{RDWRTB}. See the index entries for them.

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

In the \texttt{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 of multiple numbers, this resulting string can be broken into sub strings with the operator \texttt{"|"} via

\texttt{<string variable>|(<I1>&<I2>)}

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

\texttt{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 \texttt{DAREA} (see index), and graphics meta files can be read with the procedure \texttt{GRREA} (see index). Taylor Models can be read with the procedure \texttt{RDREA} (see index), and the binary input and output can be done with the procedures \texttt{RDREAB} and \texttt{RDWRTB} as mentioned above.

In the \texttt{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 \texttt{SF} and \texttt{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 \texttt{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 \texttt{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 6.2.

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

\texttt{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

\texttt{ST (<real variable>)}

which uses the Fortran format G23.16.

Both \texttt{SF} and \texttt{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

\texttt{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

\texttt{INCLUDE <name> ;}

includes the previously compiled code. The name follows the same syntax as in the \texttt{SAVE} command.

Each code may contain only one \texttt{INCLUDE} statement, and it has to be located at the very top of the file. The \texttt{SAVE} and \texttt{INCLUDE} statements allow breaking the code into a chain of easily manageable pieces and decrease compilation times considerably.
3.7 Parallel Environments Based on MPI

COSY Infinity provides for native routines that interface with MPI to provide parallel computations. This is useful for machines with multiple cores, or for computation on clusters. At this point, COSY has been successfully run on up to 2048 processors on the NERSC cluster in Berkeley, as well as various smaller clusters at ANL and MSU.

There are different machine and cluster specific commands that can be run, but we will reference OpenMPI calls. The user can use appropriate commands to replace their functionality. It is required that the user have access to the commands 'mpirun' and 'mpif77'.

3.7.1 Compiling Parallel COSY

It is required to have correctly prepared the source files, foxfit.f, foxgraf.f, foxy.f and dafox.f, for the MPI version of COSY. The regular COSY source can be download from COSY distribution site. Then one needs to activate the MPI supports by converting these files accordingly using the file version.f. These files can be prepared on any machine, and then transferred them to the cluster machine.

The file foxgraf.f must be stripped of all graphics output by converting *PGP to a blank string, foxy.f and dafox.f must be converted from *NORM to the *MPI option, and foxfit.f can remain the same. This can be accomplished by compiling and running version.f and then copying the newly created files to overwrite the old ones.

Next, you must compile the four files together with the appropriate command line compiler options. This should be done with the compiler wrapper function ’mpif77’ which we recommend having made with the Intel Fortran Compiler. If you are attempting to do verified computations, we highly recommend that you contact us first before proceeding. In any case, to compile to obtain a non-verified working version of the parallel COSY: mpif77 can be used in the Makefile as the Fortran Compiler instead of the usual Fortran compiler.

3.7.2 Routines

PLOOP <name> <expression> <expression> ;

and

ENDPLOOP <name> ;

Much like the LOOP construct, the first entry is the name of a visible variable which will act as the loop variable, and the first and second expressions are the first and second bounds of the loop variable. This loop construct requires that the user run through all of the processes that were asked for. i.e. if the user requests N processes when starting the parallel code, the loop must traverse each of those N processes. In almost every case the first expression will be 1 and the second expression will be the number of processes asked for. Note that it is recommended that you avoid nesting this construct. In the situation that the process is run with a single process, PLOOP behaves like the LOOP construct.

The ENDPLOOP construct takes the name associated with a visible array variable which can be used to share information between processes. e.g.
VARIABLE X 1 NP;
PLOOP I 1 NP;
{ user code };
ENDPLOOP X;

In which we share the information in the array X.

PNPRO <expression>:

This is a utility procedure that allows your program to know the number of processes. Usually it is a good idea to call this at the beginning of a program and to not hardcode in any specific number of processes. The procedure expects a visible variable in which it will insert the number of processes.

PROOT <expression>:

This is a utility procedure that expects a visible variable in which it will return a 1 if the current process is the root process.

PWTIME <expression>:

This is a utility procedure that expects a visible variable in which it will return the wall-clock time in seconds on the local node in parallel execution.

3.7.3 Running Parallel COSY

The command ‘mpirun’ can be called to run this code with a specified number of processes. e.g. mpirun -n NP ./cosy filename in which NP is an integer with the number of requested processes, ./cosy is the location of your compiled parallel cosy executable, and filename is the name of the .fox file to run.

On machines and clusters with time allocations it is suggested that the PWTIME command is used to help keep track of CPU hours.

3.8 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.
3.8 Error Messages

$$$: 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 hopefully rare case that such an error message is encountered, the user is kindly asked to contact us and submit the respective 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 $-1.0$. 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 The C++ Interface

The COSY INFINITY language environment offers an object oriented approach to advanced numerical data types. The C++ interface to COSY INFINITY (and also the F90 interface discussed in section 5) allow the use of these data types in a modern object-oriented language while retaining 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 comes close to the performance of COSY INFINITY itself and exceeds that of other approaches (the CPU time lies within a factor of two to that of the use in the COSYScript language environment on most machines).

The COSY INFINITY language (c.f. section 3) 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.

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 under-utilize 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.

4.1 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 [10]. 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. While this version may be not as up-to-date as the one available from other sources, it will always be guaranteed to work with COSY. 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 file

*.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.

### 4.2 Memory Management

The Cosy class manages its own internal memory and does not use dynamic allocation of memory by either malloc or new. In addition to the specialized numerical algorithms used for COSY’s internal data types, this fact contributes to 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.

### 4.3 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 section 3. Therefore, reading this section is essential for the understanding of the C++ interface to COSY INFINITY.

#### 4.3.1 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. The default constructor

```
Cosy( );
```

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 );
```
creates a Cosy object with enough internal space to hold \( \text{len} \) numbers or characters. The parameter \( \text{len} \) is optional and defaults to 1. The object’s type is initialized to \texttt{RE} with value \( \text{val} \).

\[
\text{Cosy( const int val, int len = 1 );}
\]

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

\[
\text{Cosy( const bool f );}
\]

creates a Cosy object with enough internal space to store one number or character. The object’s type is initialized to \texttt{LO} and its value is set to the boolean value \( f \).

\[
\text{Cosy( const char *str );}
\]

creates a Cosy object from a C string \( \text{str} \). The object’s type is set to \texttt{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 \( \text{str} \).

\[
\text{Cosy( const Cosy& src );}
\]

creates a new Cosy object from an existing one. The new object is initialized with a deep copy of \( \text{src} \). The special constructor

\[
\text{Cosy( integer len, const int n, const int dim[] );}
\]

creates a Cosy object that represents a Cosy array of dimensionality \( n \). The length of each of the dimensions is given in the array \( \text{dim} \). And each entry of the array has internal space for \( \text{len} \) numbers and is initialized to zero with type \texttt{RE}. For further details on Cosy arrays, refer to section 4.6.

### 4.3.2 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.

\[
\text{Cosy& operator =}(\text{const Cosy& rhs})
\]

assigns a deep copy of \( \text{rhs} \) to the object and return a reference to it.

\[
\text{Cosy& operator+=}(\text{const Cosy& rhs})
\]

adds \( \text{rhs} \) to the object and return a reference to it; equivalent to \( x = x + \text{rhs} \).

\[
\text{Cosy& operator-=}(\text{const Cosy& rhs})
\]

subtracts \( \text{rhs} \) from the object and return a reference to it; equivalent to \( x = x - \text{rhs} \).

\[
\text{Cosy& operator*=(const Cosy& rhs)}
\]

multiplies the object with \( \text{rhs} \) and return a reference to it; equivalent to \( x = x * \text{rhs} \).

\[
\text{Cosy& operator/=(const Cosy& rhs)}
\]

divides the object by \( \text{rhs} \) and return a reference to it; equivalent to \( x = x/\text{rhs} \).
4.3 Public Interface of the Cosy Class

Cosy& operator&=(const Cosy& rhs)

unites 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++.

4.3.3 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. The operator

Cosy operator+()

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

Cosy operator-()

returns the negative of the object without modifying it.

Cosy operator++()

adds one to the object and return the result.

Cosy operator--()

subtracts one from the object and return the result.

Cosy operator++(int)

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

Cosy operator--(int)

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

4.3.4 Array Access

In order to access COSY array elements, the command

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

obtains a copy of an array element. The element is described by the n-dimensional array coeff. More details on Cosy arrays are provided in section 4.6.

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

copies 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 section 4.6.

4.3.5 Printing, IO, and Streams

As indicated earlier, the code for the Cosy class is automatically derived from Fortran 77 code by using the F2C converter [10]. 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. The command

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

prints a representation of the object src onto the ostream s. The printing uses the formats specified in section 3.

### 4.3.6 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. The command

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

returns a double precision variable that represents the result of calling the function CONS (c.f. section 3) on the Cosy object arg.

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

returns 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.

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

returns 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.

### 4.4 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). Furthermore, all elementary type generators in COSY (the first set of intrinsic functions having two letter names) are fully capitalized, which allows for them to be clearly distinguished from other C++ tools and is inconsequential because they are only relevant for Cosy objects. A complete list of all functions supported in C++ and their explicit upper/lowercase names can be found in the file cosy.h that is part of the C++ distribution.

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 operators &, |, and % (the Cosy operators for union, Extraction, and Derivation) do not follow the standard C++ conventions. However, since the Cosy class is meant to be used for the development of new programs, or as a replacement for double variables, overloading these operators is unlikely to cause any problems.

All operators and functions listed in the appendix are available in C++, and have the following signature

\[
\text{inline <type> <name | operator op>}
\]

\[
\text{(const Cosy& lhs, const Cosy& rhs) ;}
\]

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

\[
\begin{align*}
\text{Cosy operator+} \\
\text{Cosy operator-} \\
\text{Cosy operator*} \\
\text{Cosy operator/} \\
\text{Cosy pow} \\
\text{bool operator<} \\
\text{bool operator>} \\
\text{bool operator==} \\
\text{bool operator!=} \\
\text{Cosy operator&} \\
\text{Cosy operator|} \\
\text{Cosy operator%} \\
\text{bool operator<=} \\
\text{bool operator=>}
\end{align*}
\]

The standard functions defined for the Cosy class are listed in the appendix A. 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 about their meaning, the corresponding COSY INFINITY functions should be looked up in Appendix A.

The signature of the Cosy function \texttt{¡NAME\textunderscore i} is as follows:

\[
\text{Cosy <name> ( const Cosy& x ) ;}
\]

where \texttt{¡NAME\textunderscore i} is the name of the function from the appendix. Note that for C++ use, all names have to replaced by lowercase.

### 4.5 COSY Procedures

The COSY INFINITY language environment has various intrinsic 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 complete list of the COSY procedures available in this way can be found in the appendix A. Note that a “c” parameter stands for “const Cosy &” arguments; a “v” parameter denotes “Cosy &” arguments). Note that again, the name of the procedure has to be supplied in lowercase.

### 4.6 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

\[
\text{Cosy get(const int coeff[], const int n)}
\]

and

\[
\text{void set(const Cosy& arg, const int coeff[], const int n)}
\]

described in section 4.3.4 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

\[
\begin{align*}
\text{Cosy get(Cosy &a, int i, int j) } & \{ \\
& \quad \text{int c[2] = \{i+1, j+1\}}; \\
& \quad \text{return a.get(c, 2);} \\
& \}
\end{align*}
\]

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.
5 The Fortran 90 Interface

The Fortran 90 interface to COSY INFINITY 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.

5.1 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. section 1.5 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 been 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.

5.2 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. The routine

\[
\text{SUBROUTINE COSY_INIT [<NTEMP>] [<NSCR>] [<MEMDBG>]} \]

initializes 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 50000 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 section 5.7 for more details on the pool of temporary objects.) The command

\textbf{SUBROUTINE COSY\_CREATE <SELF> [<LEN>] [<VAL>] [<NDIMS>] [<DIMS>]}\textbf{ }

creates 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 \textbf{RE}.

This routine can also be used for the creation of COSY arrays (see also section 5.8). 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 \textbf{RE}.

\textbf{SUBROUTINE COSY\_DESTROY <SELF>}

destructs the COSY object SELF and free the associated memory. If SELF hasn’t been initialized with \textbf{COSY\_CREATE}, the results of this are undefined.

\textbf{SUBROUTINE COSY\_ARRAYGET <SELF> <NDIMS> <IDXS>}

returns 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 section 5.8 for further details on COSY arrays).

\textbf{SUBROUTINE COSY\_ARRAYSET <SELF> <NDIMS> <IDXS> <ARG>}

copies 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 section 5.8 for further details on COSY arrays).

\textbf{SUBROUTINE COSY\_GETTEMP <SELF>}

returns 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 \textbf{RE}. Refer to section 5.7 for more details.

\textbf{SUBROUTINE COSY\_DOUBLE <SELF>}

extracts the DOUBLE PRECISION value from the variable SELF by calling the function COSY function \textbf{CONS}.

\textbf{SUBROUTINE COSY\_LOGICAL <SELF>}

extracts the logical value from the variable SELF. If the type of SELF is not \textbf{LO}, the result of the operation is undefined.

\textbf{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 \textbf{WRITE} (c.f. section 3.6).

\textbf{SUBROUTINE COSY\_TMP <ARG>
returns 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.

```fortran
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)
```

## 5.3 Operations

The Fortran 90 interface to COSY INFINITY offers all operators that the standard COSY system offers. For the convenience 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.

<table>
<thead>
<tr>
<th>Addition (+)</th>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSY</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Subtraction (-)</th>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSY</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>COSY</td>
<td></td>
<td>COSY</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Multiplication (*)</th>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSY</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
<td></td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
<td></td>
</tr>
</tbody>
</table>
### Division /

<table>
<thead>
<tr>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
</tr>
</tbody>
</table>

### Power **

<table>
<thead>
<tr>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
</tr>
</tbody>
</table>

### Comparison .LT.

<table>
<thead>
<tr>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
</tr>
</tbody>
</table>

### Comparison .GT.

<table>
<thead>
<tr>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
</tr>
</tbody>
</table>

### Comparison .EQ.

<table>
<thead>
<tr>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
</tr>
</tbody>
</table>

### Comparison .NE.

<table>
<thead>
<tr>
<th>COSY</th>
<th>COSY</th>
<th>COSY</th>
</tr>
</thead>
<tbody>
<tr>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>DOUBLE PRECISION</td>
<td>COSY</td>
</tr>
<tr>
<td>INTEGER</td>
<td>COSY</td>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
<td>INTEGER</td>
<td>COSY</td>
</tr>
</tbody>
</table>
5.4 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. The command

\[
\text{COSY LHS = COSY RHS}
\]

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

\[
\text{DOUBLE PRECISION LHS = COSY RHS}
\]

converts the COSY object RHS to the DOUBLE PRECISION number LHS by calling the function \text{COSY.DOUBLE}.

\[
\text{LOGICAL LHS = COSY RHS}
\]

converts the COSY object RHS to the LOGICAL variable LHS by calling the function \text{COSY.LOGICAL}.

\[
\text{COSY LHS = DOUBLE PRECISION RHS}
\]

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 \text{RE}.

\[
\text{COSY LHS = LOGICAL RHS}
\]

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 \text{LO}.

\[
\text{COSY LHS = INTEGER RHS}
\]

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 \text{RE}. 

---

<table>
<thead>
<tr>
<th>Concatenation .UN.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSY</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>COSY</td>
</tr>
<tr>
<td>INTEGER</td>
</tr>
<tr>
<td>COSY</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>INTEGER</td>
</tr>
<tr>
<td>INTEGER</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extraction .EX.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Derivation .DI.</th>
</tr>
</thead>
<tbody>
<tr>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
</tr>
<tr>
<td>COSY</td>
</tr>
</tbody>
</table>
5.5 Functions

The Fortran 90 interface to COSY INFINITY supports most of the functions supported by the COSY environment; for the few functions not supported, a compiler error message will result. Appendix A lists details on the COSY INFINITY functions.

5.6 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.

5.7 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 allocated 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 for utmost computational efficiency (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.

5.8 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> <IDX S>

and

SUBROUTINE COSY ARRAYSET <SELF> <NDIMS> <IDX S> <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.


6 Optimization, Graphics, and the Graphical User Interface

6.1 Optimization

Many design problems 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 6.1.1. Because of a relatively simple interface, it is also possible to include new optimizers relatively easily. Details can be found in section 6.1.2.

Besides the Fortran algorithms for nonlinear optimization, COSYScript allows the user to design his own problem-dependent optimization strategies because of the availability of the FIT command as a language element and the ability to nest with other control elements of the COSYScript language.

6.1.1 Optimizers

The FIT and ENDFIT commands of COSY allow the use of various different optimizers supplied in Fortran. The optimizers attempt to find optimal solutions to the problem

\[ f_i(\vec{x}) = 0, \]

where \( \vec{x} \) is a vector of \( N_v \) variables listed in the FIT command, and the \( f_i \) are \( N_f \) objectives listed in the ENDFIT command. For details on the syntax of the commands, including termination criteria and control parameters for selection of algorithms, we refer to page 22.

At the present time, COSY internally supports three different optimizers with different features and strengths and weaknesses to attempt to find optimal solutions of \( f_i = 0 \). In addition, there is the rather sophisticated rigorous global optimizer COSY-GO, but this tool can currently not be called from within the FIT - ENDFIT structure, but has as a standalone interface. In the following we present a list of the various currently supported 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. In particular, it tolerates well the use of non-smooth penalty functions, for example to restrict the search domain. It is quite rugged and finds local (and often global) minima in a rather large class of cases. In simple smooth cases, it often requires more execution time than the LMDIF algorithm. However, because of its generality at reasonable execution cost, it is often the algorithm of choice.

2. Not currently available; rerouted to “4. The LMDIF optimizer”.

3. The Simulated Annealing Algorithm
   This algorithm, a special type of the wide class of stochastic methods, attempts to find the global optimum, and often succeeds even for cases where other optimizers fail. This comes at the expense of a frequently very high and sometimes prohibitive number of function evaluations. Often this algorithm is also helpful for finding promising starting values for the subsequent use of other algorithms.

4. The LMDIF optimizer
   This optimizer is a generalized least squares Newton method with various stability enhancements,
and is very efficient in the proximity of the solution and if the objectives are smooth, but it is not as robust as the either the simplex or simulated annealing algorithms. For most cases, it should be the first optimizer to try.

It should be stressed that the success or failure of non-verified optimization tasks often rests on the clever use of strategies combining different optimizers, random search, or structured search. The COSY approach of offering the FIT - ENDFIT environment at the language level attempts to give the demanding user far-reaching freedom to tailor his own optimization strategy. This can be achieved by properly nested structures involving loops, while blocks, and if blocks in combination with the fit blocks.

### 6.1.2 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.f, which is the routine that is called from the code executor in foxy.f. The arguments for the routine are as follows:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IFIT</td>
<td>identification number of optimizer</td>
</tr>
<tr>
<td>XV</td>
<td>current array of variables</td>
</tr>
<tr>
<td>NV</td>
<td>number of variables</td>
</tr>
<tr>
<td>EPS</td>
<td>desired accuracy of function value</td>
</tr>
<tr>
<td>ITER</td>
<td>maximum allowed iteration number</td>
</tr>
<tr>
<td>IEND</td>
<td>status identifier</td>
</tr>
</tbody>
</table>

The last argument, the status identifier, communicates the status of the optimization process to the executor 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.f in order to be able to distribute the optimizer to other users as well.
6.2 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 55.

6.2.1 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:

\texttt{FG \textless PIC \textgreater XL \textless XR \textless YB \textless YT \textless DX \textless DY \textless TITLE \textless I \textgreater ;}

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:

\texttt{CG \textless PIC \textgreater X \textless Y \textless N \textgreater ;}

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 \texttt{FG} before calling this routine.

6.2.2 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 COSYScript using COSY’s \texttt{WRITE} command. The different unit numbers correspond to the following drivers:

- positive: Low-Resolution ASCII output to respective unit; 6: screen.
- -1 ... -9: Standard interactive window output
- -10: Direct PostScript output to files pic001.ps, pic002.ps, ... The header of each PostScript file contains the useful information such as how to include the picture in a \LaTeX{} document.
- -11: Direct output to the low level graphics meta files mpic001.dat, mpic002.dat, ...
- -12: Direct \LaTeX{} picture mode output to files lpic001.tex, lpic002.tex, ...
- -20: PGPLOT output to PostScript files pgpic001.ps, pgpic002.ps, ...
- -22: PGPLOT output to \LaTeX{} files pgpic001.tex, pgpic002.tex, ...
- -101 ... -110: PGPLOT X-Windows workstation or Windows PC windows output
6.2 Graphics

-111 ... -120: GrWin  Microsoft Windows PC windows output
-121 ... -130: AquaTerm  Mac PC windows output

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.

Standard interactive window output (-1 ... -9 ) uses GrWin for Microsoft Windows PC, AquaTerm for Mac PC, and PGPLOT when both GrWin and AquaTerm are not available but PGPLOT is available.

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

-101 ... -110, -20, -22: PGPLOT package
The PGPLOT Graphics Library is freely available for download from the PGPLOT web page, which is located at http://astro.caltech.edu/~tjp/pgplot/. Download and install the library 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 library.
If linking to PGPLOT package is not desired, the PGPLOT driver routines in foxgraf.f have be modified by commenting all lines that contain the string *PGP in columns 73 to 80.

-111 ... -120: GrWin package
The GrWin Graphics Library is freely available for download from the GrWin web page, which is located at http://spdg1.sci.shizuoka.ac.jp/grwinlib/english.
If linking to GrWin package is desired, the GrWin driver routines in foxgraf.f have be modified by un-commenting all lines that contain the string *GRW in columns 73 to 80.

-121 ... -130: AquaTerm package
If linking to AquaTerm package is desired, the AquaTerm driver routines in foxgraf.f have be modified by un-commenting all lines that contain the string *AQT in columns 73 to 80.

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

\texttt{GRREAD <unit> <PIC> ;}

6.2.3 Adding Graphics Drivers

To facilitate the adaptation to new graphics packages, COSY INFINITY has a very simple standardized graphics interface in the file foxgraf.f. 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(ST,L) : Prints ASCII string ST with length L to momentary position.

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(W) : Sets the width of the pen. W ranges from 0 to 10, 0 denoting the finest, 1 the default thickness, 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, Y and W are DOUBLE PRECISION, and I is INTEGER. After these routines have been created, the routine GRPRI in foxgraf.f 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.

6.2.4 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 Gnuplot or by programs written by the user.

The meta file consists of a list of elementary operations discussed in the last subsection. Each occurrence 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 in the Fortran format 2E24.16, the character A in its actual length, and the integer I in the Fortran format I1.

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

6.3 Graphical User Interface

Starting with version 9.1, COSY INFINITY has built in support for building graphical user interfaces (GUIs) from within COSY SCRIPT. The GUI feature requires COSY to be run with a GUI program, such as the platform independent Java GUI program “COSYGUI.jar” distributed with COSY INFINITY (see Section 6.3.6).

The programming of GUI interfaces from within COSY fits in naturally with the classic way of handling input and output, while providing a wide range of commonly used GUI elements. COSY provides the
special GUI unit numbers -201 …-210, each of which represents one window in the user’s graphical environment.

Existing programs can be easily converted to use a GUI (see Section 6.3.1) with only minimal modifications to existing code. For more sophisticated GUIs, a variety of special GUI commands can be written to the GUI unit numbers to define the elements in each window and to interact with them. Available commands are described in detail in Section 6.3.3 below.

The GUI also allows input from and output to the traditional console units 5 and 6 in a GUI program. These calls are automatically routed to a separate terminal window if COSY is run in a GUI environment. Similarly, a simple ASCII based GUI is shown instead of a GUI window when COSY is run in a non-GUI environment (e.g. from the command line).

6.3.1 Basic GUIs

The main conceptual difference between a GUI and the traditional console based I/O is the concept of a delayed read. In a GUI window, the user can enter values into various fields and modify them in any order before pushing a button, which then causes all values to be read in at once.

This concept is integrated into COSY by making READ commands to the GUI window units delayed. That means that COSY will not immediately read a value and place it into the variable passed to the READ command. Instead, COSY will associate each variable with a GUI input field, and only place values in them once a delayed read is initiated. At what point in the code such a delayed read is to be performed, is up to the programmer to specify.

To convert an existing program using traditional console based I/O to a GUI program, the developer generally has to perform the following steps:

1. Change WRITE 6 to WRITE -201 to output to a GUI window instead of the terminal.

2. Similarly, change READ 5 to READ -201 to read input from a GUI window instead of the terminal. Note that the GUI unit number is the same for both READ and WRITE commands.

3. Insert the call GUIIO -201; at the correct places where you want to initiate a delayed read from the window. In this form, the command will automatically add an OK button at the end of the window, show it to the user, wait for the button to be pushed, and then fill in the values from each input field into the variables specified in the READ calls.

The WRITE commands to the GUI units will output each string as a line of simple text in the GUI, while all other data types will appear in an embedded console in the GUI window the same way they would appear in a terminal. The user can select and copy content out of an embedded console, and scroll if the content is too long. Consecutive output into a console is appended to an existing console until a string is written to the window.

For each READ from a GUI window unit, COSY will insert an input field in a separate line in the GUI. The variable to be read is associated with this input field, and its value is placed in the variable once the window is shown to the user by calling GUIIO.

When converting programs to use the GUI, developers must make sure that their code is ready for the delayed read concept. In particular, the variable being read cannot be used in the code before the call to GUIIO. Furthermore, all READ commands must read into different variables to be useful, otherwise the variable will only contain the value of the last READ command.
6.3.2 Advanced GUIs

For more fine grained control over the appearance of the GUI, the full GUI interface can be controlled through special GUI commands written to the GUI window units. The COSY GUI operates with double buffered windows, that is for each window number there is the currently displayed window and a second hidden window. Most GUI commands act on the hidden window, but some can be issued to manipulate the currently displayed window (if any).

In general, the code structure to define a GUI window looks very much like the traditional console based I/O code, where the user is prompted for some input through a WRITE and the input is then read form the user by a READ. In COSY’s GUI model, the GUI window is still constructed by issuing WRITE commands to prompt the user for input, immediately followed by READ commands to read back the actual input. The READs are automatically delayed by COSY until a delayed read is initiated.

GUI commands are issued by writing to the corresponding GUI window output unit using WRITE. GUI commands are strings starting with the backslash character (\), e.g. \ReadField. Each GUI command can take a number of arguments. Those are specified as additional arguments to the WRITE call. Their type can be anything COSY can convert into a string using the ST function. A single WRITE command may contain several GUI commands.

To read back a value from a GUI element, a READ command is issued to a GUI window unit. This associates the variable given to the READ command with the most recently written GUI field that can return a value, provided it has not been associated yet. If there is no such field, either because no GUI field that returns a value has been written yet or because the last GUI field has been associated (“read”) already, the READ command will instead insert a new \ReadField field on its own line into the GUI window, and associate the variable with that field.

To initiate the delayed read into these associated variables, the command GUIIO is used. It can be used in two different ways, depending on how it is called:

GUIIO <unit> ;

If called with only one argument, <unit> specifies the GUI window unit to read from. The command adds an OK button at the end of the window if no button was defined yet, shows the window, waits for a button to be pushed, reads all values from the window, and then closes the window.

GUIIO <unit> <button> ;

If called with two arguments, <unit> specifies the GUI window unit, and <button> must be a variable to receive the name of the button that was pushed. In this more advanced form, GUIIO only waits for a button to be pushed in the currently displayed window, and then reads the values of all associated variables. The text on the button that was pushed is stored in <button> (note that this string is subject to COSY’s usual READ processing). It does not modify the window in any way (e.g. showing it, adding buttons, or closing it). If there is no window currently displayed, all variables are filled with zeros immediately and the number –1 is returned in <button>. If there is a window displayed, but it does not have a button, all variables are read immediately and the number 0 is returned in <button>.

6.3.3 GUI Command Reference

Table 1 lists all available GUI commands currently implemented in COSY. The first column gives the name of the command. Commands are case insensitive, the spelling used here is by convention but not required. Commands starting with ”Read” insert a GUI element that can be read by a subsequent READ call. The second column specifies which of the two windows the command acts on (either hidden or currently
6.3 Graphical User Interface

<table>
<thead>
<tr>
<th>Command</th>
<th>Window</th>
<th>Value</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\Console</code>: Write to embedded console</td>
<td>hidden</td>
<td>No</td>
<td>Any number of arguments of any type</td>
</tr>
<tr>
<td><code>\Text</code>: Static text</td>
<td>hidden</td>
<td>No</td>
<td>String to be inserted</td>
</tr>
<tr>
<td><code>\Image</code>: Static image</td>
<td>hidden</td>
<td>No</td>
<td>Image filename</td>
</tr>
<tr>
<td><code>\Line</code>: Vertical line</td>
<td>hidden</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td><code>\Spacer</code>: Transparent element</td>
<td>hidden</td>
<td>No</td>
<td>Width in pixels Height in pixels (0)</td>
</tr>
<tr>
<td><code>\Button</code>: Push button</td>
<td>hidden</td>
<td>No</td>
<td>Text on button 1 - default button, 0 - otherwise (0) Tooltip (none)</td>
</tr>
<tr>
<td><code>\ReadCheckbox</code>: Checkbox</td>
<td>hidden</td>
<td>Yes</td>
<td>Text next to checkbox (none) 1 - selected, 0 - not selected (0) Tooltip (none)</td>
</tr>
<tr>
<td><code>\ReadOption</code>: Radio button</td>
<td>hidden</td>
<td>Yes</td>
<td>Text next to radio button (none) 1 - selected, 0 - not selected (0) name of button group (none) Tooltip (none)</td>
</tr>
<tr>
<td><code>\ReadField</code>: Unformatted input field</td>
<td>hidden</td>
<td>Yes</td>
<td>Initial value (none) Tooltip (none)</td>
</tr>
<tr>
<td><code>\ReadNumber</code>: Numerical input</td>
<td>hidden</td>
<td>Yes</td>
<td>Current value Minimum value Maximum value Increment ((Max-Min)/100) 1 - editable, 0 - not editable (1) Tooltip (none)</td>
</tr>
<tr>
<td><code>\ReadList</code>: Selection from list</td>
<td>hidden</td>
<td>Yes</td>
<td>List of entries separated by</td>
</tr>
<tr>
<td><code>\ReadFileName</code>: File selector</td>
<td>hidden</td>
<td>Yes</td>
<td>Initial value (none)</td>
</tr>
<tr>
<td><code>\ReadProgress</code>: Progress bar</td>
<td>hidden</td>
<td>Yes</td>
<td>Progress in % or -1 (-1)</td>
</tr>
<tr>
<td><code>\NewLine</code>: Jump to next line</td>
<td>hidden</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td><code>\NewCell</code>: Jump to next cell</td>
<td>hidden</td>
<td>No</td>
<td>Width of cell (1)</td>
</tr>
<tr>
<td><code>\Left</code>: Set current cell’s alignment to left</td>
<td>hidden</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td><code>\Center</code>: Set current cell’s alignment to center</td>
<td>hidden</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td><code>\Right</code>: Set current cell’s alignment to right</td>
<td>hidden</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td><code>\Just</code>: Set current cell’s alignment to justified</td>
<td>hidden</td>
<td>No</td>
<td>None</td>
</tr>
<tr>
<td><code>\Title</code>: Set window title</td>
<td>hidden</td>
<td>No</td>
<td>Window title</td>
</tr>
<tr>
<td><code>\Deactivate</code>: Make non-interactive</td>
<td>displayed</td>
<td>N/A</td>
<td>None</td>
</tr>
<tr>
<td><code>\Activate</code>: Make interactive</td>
<td>displayed</td>
<td>N/A</td>
<td>None</td>
</tr>
<tr>
<td><code>\Show</code>: Display window</td>
<td>hidden</td>
<td>N/A</td>
<td>x coordinate (center) y coordinate (center)</td>
</tr>
<tr>
<td><code>\Close</code>: Close and destroy window</td>
<td>displayed</td>
<td>N/A</td>
<td>None</td>
</tr>
<tr>
<td><code>\Set</code>: Set value of interactive element</td>
<td>displayed</td>
<td>N/A</td>
<td>Number of element Value to be set</td>
</tr>
<tr>
<td><code>\Focus</code>: Make this the active window</td>
<td>displayed</td>
<td>N/A</td>
<td>None</td>
</tr>
<tr>
<td><code>\Debug</code>: Set GUI debug level</td>
<td>all</td>
<td>N/A</td>
<td>Debug level between 0 – 3</td>
</tr>
<tr>
<td><code>\Finish</code>: Add a button if none is there yet</td>
<td>hidden</td>
<td>No</td>
<td>Text on the button (’OK’)</td>
</tr>
</tbody>
</table>

Table 1: Available GUI commands in COSY
displayed). The third column indicates whether a command returns a value when the GUI is read from. The last column lists the arguments the command takes. Optional arguments are indicated by a default value in parenthesis, if they are omitted, this value is used. Optional arguments can only be omitted beginning with the last argument.

In the following we give some further remarks on specific GUI commands:

\**Console**
All arguments are output in the same form as on a regular terminal. An embedded console is inserted into the GUI window on a separate line. Output is appended to this console until another GUI component is added, or one of \NewLine, \NewCell, or \Show are called. The user can select and copy text in an embedded console, scroll if the text is too long, but cannot change the content.

\**Image**
Image file names are specified with forward slashes (/) as path separators or as file:/// URLs for full paths. Any fully qualified URL can be given to load images over the internet (if the computer has an internet connection). The Java GUI shipped with COSY INFINITY comes with some commonly used icons built in which can be accessed using URLs of the form cosy:///yes.png, where instead of "yes.png" any one of the built in icons ("ask.png", "clock.png", "cosy.png", "info.png", "msu.png", "msupa.png", "no.png", "star.png", "warn.png", "wrench.png", "yes.png") can be used.

\**ReadNumber**
When editable, this will display an input field with adjoining up and down buttons. Only numeric input is allowed in this field. When not editable, a slider is shown which can be dragged by the user to indicate a numeric value. When read, this field always returns a number in COSY.

\**ReadOption**
Options are a group of GUI elements of which only one can be selected at a time (typically displayed as round buttons). In order to designate which option belongs to which group, the name of an option group can be specified. Of all options in a group with the same name, at most one is selected at each time.

\**ReadList**
Presents the user with a list of options from which to select one. If the list is set to editable, the user is allowed to enter a value that is not on the list, otherwise the user must select a value on the list.

\**ReadProgress**
This element can either display a progress bar with the given percentage of completion, or a bar with an indeterminate state to indicate that a computation is ongoing but the total time is not known. When read, this element will simply return the value it is currently set to. This is mostly so that it can have its value changed while the window is displayed using \Set.

\NewLine, \NewCell, \Left, \Center, \Right, \Just
See Section 6.3.4 for information about the layout of GUI elements.

\**Deactivate, Activate**
These commands make the currently displayed window either inactive or active. In an inactive window, the user cannot interact with the elements of the window any more, they are often shown in gray. This command does not change window visibility, the window remains visible all the time. By default, windows are active, i.e. the user can interact with the elements in the window.

\**Show**
This command closes and destroys the currently displayed window, if any, and replaces it by the hidden window, which is made visible to the user. If no arguments are given, the new window is
shown where the previously displayed window was, otherwise it is centered on the screen. If both
coordinates are given, the window’s top left corner is positioned accordingly, with (0,0) being the
top left corner of the screen, and (1,1) the bottom right.
All subsequent calls to create GUI elements will act on a newly created, initially empty hidden
window. This command returns immediately, to wait for user input use **GUIIO**.

\Set
To update the value of a component in the currently displayed window, this command can be called.
The number of the element is determined by the order in which GUI elements were added to the
window counting only elements that return a value starting with 1.

\Debug
Set the debug level for the GUI. Integer between 0 and 3, with 0 (the default) meaning no debug
output, 1 meaning errors are logged to the Terminal, 2 also outputting diagnostic messages, and 3
echoing the entire GUI protocol read from COSY. Can be called several times to turn debugging of
certain parts of the GUI on or off.

6.3.4 GUI Layout
Components in the GUI are arranged based on the order in which they are added and a natural size for
each element as determined by the GUI program. Each element is added at the end of the current line
in the GUI. A line can be ended using the **NewLine** command, which causes all further elements to be
added at the beginning of the next line.

The size of the window is determined by the width of the longest line, and the total number of lines. If
a line is shorter than the width of the resulting window, it is aligned according to the alignment specified
by one of the commands **Left**, **Center**, **Right**, or **Just**. **Just** will cause the elements in the line to
be resized such that they fill up the entire line. By default, if none of the alignment commands was issued,
lines are left justified.

Alignment commands can be called at any time, before or after writing elements to a line. It always
applies to the current line and if called multiple times within the same line, the last call carries.

For more sophisticated layouts, the COSY GUI specification supports the **NewCell** command. With
this command it is possible to lay out elements in a tabular grid, where each cell behaves much like the
lines described above. Each cell can have its own alignment, and the size of each row and column in
the table is determined by the largest cell in the row or column. A row of cells is ended by calling the
**NewLine** command.

By providing an integer argument to **NewCell**, the current cell can be made to span multiple cells.
The last cell in each row is automatically expanded to the end of the window, so it is not necessary to
provide a cell span for the last cell. If this behavior is not desired, an empty cell can be inserted after the
last occupied cell by simply calling **NewCell**.

6.3.5 Examples

- **WRITE** -201 ' \NewLine' \NewLine \NewLine';
  Inserts three empty lines in window number -201.

- **WRITE** -201 ' \ReadNumber' tax 0 100;
  Inserts a slider with its initial value taken from variable tax and minimum value 0 and maximum
  value 100 in window number -201.
7 ACKNOWLEDGEMENTS

- WRITE -201 '\ReadField' name '\NewLine';
  Inserts an input box with initial text taken from variable name followed by a new line in window number -201.

- WRITE -201 '\ReadList' 'Ra|Zeus|Jupiter|Other' 'Zeus' 0 'Select yours!';
  Inserts a non-editable list with the options "Ra", "Zeus", "Jupiter", and "Other", with "Zeus" initially selected and a tooltip of "Select yours!" in window number -201.

- WRITE -201 '\Show'; GUIIO -201 button;
  Show window number -201 and wait for a button to be pushed in this window. The name of the button is stored in variable button.

For complete examples of GUI usage in a simple program, see gui.fox. An overview over all available GUI elements and what they look like is given in elements.fox. For an example of how to covert an existing program into a GUI program, see the brief COSY demo in briefdemo_full.fox (advanced GUI), briefdemo_basic.fox (basic GUI), and briefdemo.fox (console only), or the COSY beam physics demo in demo_full.fox (advanced GUI), demo_basic.fox (basic GUI), and demo.fox (console only).

6.3.6 Java GUI

In order to run the Java GUI program distributed with the COSY INFINITY installer packages for Windows and Mac OS X, you must at least have Java 5 installed. If you do not have Java installed already, you can get Java for free at http://www.java.com.

The typical user will run the Java GUI through the file association set up by the installer package. You should just be able to right click any FOX file, and select “Run COSY” (Windows) or “Open With → Run COSY” (Mac OS X). Alternatively, you can start the program by selecting “Run COSY” from your Start Menu (Windows) or Applications folder (Mac OS X). The Java GUI will then prompt you to navigate to a FOX file to run.

For Linux users, depending on your desktop environment, you can either start the GUI by double clicking the COSYGUIGUI.jar file, or using the command line.

The Java GUI tries to find the COSY executable to use by searching the following locations for "cosy.exe" (Windows) or "cosy" (Mac OS X, Unix) in order, using the first hit:

1. Location of the FOX file being run
2. Location of the COSYGUIGUI.jar file
3. Operating system dependent location where COSY was installed

In order to use a custom built COSY executable (e.g. because of memory requirements), one can simply copy the executable into the same directory as the FOX file being run. Then COSY can be run by the means provided by the installer on each operating system, and will automatically use the correct COSY executable.

7 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, Jens Hoefkens, Béla Erdélyi, Michael Lindemann, Ralf Tönjes, Laura Chapin, Carlos Maidana, Shashikant Manikonda, Youn-Kyung Kim, Pavel Snopok, Alexey Poklonskiy, Johannes Grote, and Alexander Wittig 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 Jorge More for providing the public domain optimizer LMDIF.

COSY INFINITY makes use of the following programs and libraries on various platforms: GRWin by Tamaribuchi, Tsuguihiro for plotting on Windows, AquaTerm for plotting on Mac OS X, PGPPlot for plotting on Linux, WeFunction icon set from www.wefunction.com for GUI icons in the COSYGUI Java program.

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

References


A  The Supported Types and Operations

Within the COSY INFINITY environment, object types and operations on them can be defined by the language description file genfox.dat. This file is read by the program GENFOX, which then updates the source code of the COSY system and updates the \LaTeX source of this manual.

The first part in genfox.dat is a list of the names of all data types. The second part 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 part contains all the intrinsic functions and the types of their results. The fourth part finally contains a list of Fortran procedures that can be called from the environment.

Below follows a GENFOX-generated list of currently available 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.

The subsequent information is automatically generated by the GENFOX syntax management system, and is current as of 20-Jun-2011.

A.1  Objects

In this version of COSY INFINITY, the following objects or data types are supported:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>8 Byte Real Number</td>
</tr>
<tr>
<td>ST</td>
<td>String</td>
</tr>
<tr>
<td>LO</td>
<td>Logical</td>
</tr>
<tr>
<td>CM</td>
<td>8 Byte Complex Number</td>
</tr>
<tr>
<td>VE</td>
<td>Vector of 8 Byte Real Numbers</td>
</tr>
<tr>
<td>IN</td>
<td>8 Byte Interval Number</td>
</tr>
<tr>
<td>IV</td>
<td>Vector of 8 Byte Intervals</td>
</tr>
<tr>
<td>DA</td>
<td>Differential Algebra Vector</td>
</tr>
<tr>
<td>CD</td>
<td>Complex Differential Algebra Vector</td>
</tr>
<tr>
<td>TM</td>
<td>Taylor Model (Remainder-enhanced DA Vector)</td>
</tr>
<tr>
<td>GR</td>
<td>Graphics</td>
</tr>
</tbody>
</table>
A.2 Operators

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

- **+ (Addition)** (Priority: 3)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td></td>
<td>CM</td>
</tr>
<tr>
<td>RE</td>
<td>CM</td>
</tr>
<tr>
<td>RE</td>
<td>VE</td>
</tr>
<tr>
<td>RE</td>
<td>IN</td>
</tr>
<tr>
<td>RE</td>
<td>IV</td>
</tr>
<tr>
<td>RE</td>
<td>DA</td>
</tr>
<tr>
<td>RE</td>
<td>CD</td>
</tr>
<tr>
<td>RE</td>
<td>TM</td>
</tr>
<tr>
<td>LO</td>
<td>LO</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>CM</td>
<td>DA</td>
</tr>
<tr>
<td>CM</td>
<td>CD</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>VE</td>
<td>IN</td>
</tr>
<tr>
<td>VE</td>
<td>IV</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IN</td>
<td>IV</td>
</tr>
<tr>
<td>IN</td>
<td>TM</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
<tr>
<td>IV</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>CM</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>DA</td>
<td>CD</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>CM</td>
</tr>
<tr>
<td>CD</td>
<td>DA</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>IN</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>
- **(Subtraction)** (Priority: 3)

<table>
<thead>
<tr>
<th>Type</th>
<th>Type</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
<td>RE</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>RE</td>
<td>CM</td>
<td>CM</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>RE</td>
<td>VE</td>
<td>VE</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>RE</td>
<td>IN</td>
<td>IN</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>RE</td>
<td>IV</td>
<td>IV</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>RE</td>
<td>DA</td>
<td>DA</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>RE</td>
<td>CD</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>RE</td>
<td>TM</td>
<td>TM</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
<td>CM</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
<td>CM</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CM</td>
<td>DA</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CM</td>
<td>CD</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
<td>VE</td>
<td>Subtract Real componentwise</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
<td>VE</td>
<td>Subtract Real componentwise</td>
</tr>
<tr>
<td>VE</td>
<td>IN</td>
<td>IV</td>
<td>Subtract Interval componentwise</td>
</tr>
<tr>
<td>VE</td>
<td>IV</td>
<td>IV</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
<td>IN</td>
<td>Subtract componentwise from Interval</td>
</tr>
<tr>
<td>IN</td>
<td>VE</td>
<td>IV</td>
<td>Subtract componentwise from Interval</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
<td>IN</td>
<td>Subtract componentwise from Interval</td>
</tr>
<tr>
<td>IN</td>
<td>IV</td>
<td>IV</td>
<td>Subtract componentwise from Interval</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
<td>IV</td>
<td>Subtract componentwise from Interval</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
<td>IV</td>
<td>Subtract componentwise from Interval</td>
</tr>
<tr>
<td>IV</td>
<td>IN</td>
<td>IV</td>
<td>Subtract Interval componentwise</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
<td>IV</td>
<td>Subtract componentwise from Interval</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
<td>DA</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>DA</td>
<td>CM</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
<td>DA</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>DA</td>
<td>CD</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CD</td>
<td>CM</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CD</td>
<td>DA</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
<td>CD</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
<td>TM</td>
<td>Subtract componentwise from Real</td>
</tr>
<tr>
<td>TM</td>
<td>IN</td>
<td>TM</td>
<td>Subtract Interval from the remainder of TM</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
<td>TM</td>
<td>Subtract Interval from the remainder of TM</td>
</tr>
</tbody>
</table>
- **Multiplication** (Priority: 4)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>Multiply with Real componentwise</td>
</tr>
<tr>
<td>CM</td>
<td>Multiply with Real componentwise</td>
</tr>
<tr>
<td>VE</td>
<td>Multiply with Real componentwise</td>
</tr>
<tr>
<td>IN</td>
<td>Multiply with Interval componentwise</td>
</tr>
<tr>
<td>IV</td>
<td>Multiply with Interval componentwise</td>
</tr>
<tr>
<td>DA</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>CD</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>LO</td>
<td>Logical AND</td>
</tr>
<tr>
<td>CM</td>
<td></td>
</tr>
<tr>
<td>CM</td>
<td></td>
</tr>
<tr>
<td>CM</td>
<td></td>
</tr>
<tr>
<td>CM</td>
<td></td>
</tr>
<tr>
<td>CM</td>
<td></td>
</tr>
</tbody>
</table>

- **Multiplication** (Priority: 4)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>VE</td>
<td>Multiply with Real componentwise</td>
</tr>
<tr>
<td>VE</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>IN</td>
<td>Multiply with Interval componentwise</td>
</tr>
<tr>
<td>IV</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td></td>
</tr>
<tr>
<td>IV</td>
<td>Multiply with Interval componentwise</td>
</tr>
<tr>
<td>IV</td>
<td>Multiply with Interval componentwise</td>
</tr>
<tr>
<td>IV</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>DA</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>DA</td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td></td>
</tr>
<tr>
<td>DA</td>
<td></td>
</tr>
<tr>
<td>CD</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>CD</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>CD</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>CD</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>TM</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>TM</td>
<td>Multiply componentwise</td>
</tr>
<tr>
<td>TM</td>
<td>Multiply componentwise</td>
</tr>
</tbody>
</table>
### A.2 Operators

- **/ (Division)** (Priority: 4)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>RE</td>
<td>CM</td>
</tr>
<tr>
<td>RE</td>
<td>VE</td>
</tr>
<tr>
<td>RE</td>
<td>IN</td>
</tr>
<tr>
<td>RE</td>
<td>IV</td>
</tr>
<tr>
<td>RE</td>
<td>DA</td>
</tr>
<tr>
<td>RE</td>
<td>CD</td>
</tr>
<tr>
<td>RE</td>
<td>TM</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>CM</td>
<td>DA</td>
</tr>
<tr>
<td>CM</td>
<td>CD</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>VE</td>
<td>IN</td>
</tr>
<tr>
<td>VE</td>
<td>IV</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IV</td>
</tr>
<tr>
<td>IN</td>
<td>IV</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
<tr>
<td>IV</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>CM</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>DA</td>
<td>CD</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>CM</td>
</tr>
<tr>
<td>CD</td>
<td>DA</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **^ (Exponentiation)** (Priority: 5)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
</tbody>
</table>

Raise to Real power componentwise
- **< (Less Than) (Priority: 2)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **> (Greater Than) (Priority: 2)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **= (Equal) (Priority: 2)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
</tbody>
</table>

- **# (Not Equal) (Priority: 2)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
</tbody>
</table>

- **& (Concatenation) (Priority: 2)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>RE</td>
<td>VE</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IN</td>
<td>IV</td>
</tr>
<tr>
<td>IV</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>GR</td>
<td>GR</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### A.2 Operators

- **| (Extraction) (Priority: 6)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>ST</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>VE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>VE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>VE</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>VE</td>
</tr>
</tbody>
</table>

- **% (Derivation) (Priority: 7)**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Left</td>
<td>Right</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
</tr>
</tbody>
</table>
A.3 Intrinsic Functions

The following is a list of all available intrinsic functions. Each function has a single argument. Also shown are all allowed incoming types and the resulting types of the function.

- **RE**  Converts various types to Real (RE)

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>RE</td>
<td>RE</td>
<td>(no effect)</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
<td>RE</td>
<td>Determines the average</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
<td>RE</td>
<td>Determines midpoint of interval</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
<td>RE</td>
<td>Extracts constant part of DA</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
<td>RE</td>
<td>Extracts constant part of TM</td>
</tr>
</tbody>
</table>

- **ST**  Converts various types to String (ST)

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>ST</td>
<td>ST</td>
<td>Formatted Conversion</td>
</tr>
<tr>
<td>ST</td>
<td>ST</td>
<td>ST</td>
<td>(no effect)</td>
</tr>
<tr>
<td>LO</td>
<td>ST</td>
<td>ST</td>
<td>Text of the logical values True or False</td>
</tr>
<tr>
<td>CM</td>
<td>ST</td>
<td>ST</td>
<td>Formatted Conversion</td>
</tr>
<tr>
<td>IN</td>
<td>ST</td>
<td>ST</td>
<td>Formatted Conversion</td>
</tr>
</tbody>
</table>

- **LO**  Converts various types to Logical (LO)

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>LO</td>
<td>LO</td>
<td>1: True, 0: False</td>
</tr>
<tr>
<td>LO</td>
<td>LO</td>
<td>LO</td>
<td>(no effect)</td>
</tr>
</tbody>
</table>

- **CM**  Converts various types to Complex (CM)

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Result</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM</td>
<td>CM</td>
<td>CM</td>
<td>(no effect)</td>
</tr>
<tr>
<td>VE</td>
<td>CM</td>
<td>CM</td>
<td>Converts two-vector with real and imaginary parts</td>
</tr>
<tr>
<td>CD</td>
<td>CM</td>
<td>CM</td>
<td>Extracts constant part from Complex DA Vector</td>
</tr>
</tbody>
</table>
### A.3 Intrinsic Functions

- **VE** Converts various types to Vector (VE)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>(no effect)</td>
</tr>
<tr>
<td>CM</td>
<td>VE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>VE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
</tbody>
</table>

- **IN** Converts various types to Interval (IN)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>IN</td>
</tr>
<tr>
<td>VE</td>
<td>IN</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IN</td>
</tr>
<tr>
<td>TM</td>
<td>IN</td>
</tr>
</tbody>
</table>

- **IV** Converts various types to Interval Vector (IV)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>IN</td>
</tr>
<tr>
<td>VE</td>
<td>IV</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>TM</td>
<td>IV</td>
</tr>
</tbody>
</table>

- **DA** Converts various types to DA Vector

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>DA</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>CD</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>DA</td>
</tr>
</tbody>
</table>

- **CD** Converts various types to Complex DA Vector (CD)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>CD</td>
</tr>
<tr>
<td>DA</td>
<td>CD</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
</tr>
</tbody>
</table>
• **TM** Converts various types to Taylor Model (TM)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>Generates i-th component of identity TM on domain [-1,1]</td>
</tr>
<tr>
<td>IN</td>
<td>Produces a TM with zero polynomial and remainder</td>
</tr>
<tr>
<td>TM</td>
<td>(no effect)</td>
</tr>
</tbody>
</table>

• **LRE** Determines allocation size of Real (RE)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>

• **LST** Determines allocation size of String (ST)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE [Input: length of string]</td>
</tr>
</tbody>
</table>

• **LLO** Determines allocation size of Logical (LO)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE [Input: arbitrary]</td>
</tr>
</tbody>
</table>

• **LCM** Determines allocation size of Complex (CM)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE [Input: arbitrary]</td>
</tr>
</tbody>
</table>

• **LVE** Determines allocation size of Vector (VE)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE [Input: number of components]</td>
</tr>
</tbody>
</table>

• **LIN** Determines allocation size of Interval (IN)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE [Input: arbitrary]</td>
</tr>
</tbody>
</table>

• **LIV** Determines allocation size of Interval Vector (IV)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE [Input: number of components]</td>
</tr>
</tbody>
</table>
A.3 Intrinsic Functions

- **LDA** Determines allocation size of DA

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **LCD** Determines allocation size of Complex DA Vector (CD)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **LTM** Determines allocation size of Taylor Model (TM)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **LGR** Determines allocation size of Graphics (GR)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **TYPE** Returns the type of an object as a number in internal order

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>RE</td>
</tr>
<tr>
<td>LO</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
</tr>
<tr>
<td>GR</td>
<td>RE</td>
</tr>
</tbody>
</table>
- **LENGTH** Returns the currently used memory of an object (8 byte blocks)

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>LO</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>GR</td>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **VARMEM** Returns the current memory address of an object

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>LO</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>GR</td>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **VARPOI** Returns the current pointer address of an object

<table>
<thead>
<tr>
<th>Type</th>
<th>Argument</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>ST</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>LO</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>GR</td>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>
- **ARI** Returns the status for verified types (0:normal, 1:abnormal)

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>RE</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **EXP** Computes the exponential function

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **LOG** Computes the natural logarithm

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **SIN** Computes the sine

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>
- **COS** Computes the cosine

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **TAN** Computes the tangent

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **ASIN** Computes the arc sine

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **ACOS** Computes the arc cosine

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>


- **ATAN** Computes the arc tangent

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **SINH** Computes the hyperbolic sine

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **COSH** Computes the hyperbolic cosine

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **TANH** Computes the hyperbolic tangent

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>
- **SQRT** Computes the square root

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **ISRT** Computes the reciprocal of the square root, $x^{-1/2}$

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **ISRT3** Computes the reciprocal to the power $3/2$, $x^{-3/2}$

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **SQR** Computes the square

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>
### A.3 Intrinsic Functions

- **ERF** Computes the real error function $\text{erf}$

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
</tbody>
</table>

- **WERF** Computes the complex error function $w$

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
</tr>
</tbody>
</table>

- **VMIN** Computes the minimum of vector elements

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **VMAX** Computes the maximum of vector elements

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **ABS** Computes the absolute value

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **NORM** Computes the norm of a vector

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>VE</td>
<td>VE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>RE</td>
</tr>
</tbody>
</table>
- **CONS** Determines the constant part of certain types

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>VE</td>
<td>RE</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
<tr>
<td>DA</td>
<td>RE</td>
</tr>
<tr>
<td>CD</td>
<td>CM</td>
</tr>
<tr>
<td>TM</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **LDB** Computes the interval bound of a TM using LDB

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>TM</td>
<td>IN</td>
</tr>
</tbody>
</table>

- **INL** Returns the lower bound of certain interval types

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
</tbody>
</table>

- **INU** Returns the upper bound of certain interval types

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
</tbody>
</table>

- **WIDTH** Computes the (approximate) width of an interval

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
</tbody>
</table>

- **IWIDTH** Computes a rigorous enclosure of the width of an interval

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
</tbody>
</table>
### A.3 Intrinsic Functions

- **VOLUM** Computes the (approximate) volume of an interval box
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>IN</td>
<td>RE</td>
</tr>
<tr>
<td>IV</td>
<td>VE</td>
</tr>
</tbody>
</table>

- **IVOLUM** Computes a rigorous enclosure of the volume of an interval box
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>IN</td>
<td>IN</td>
</tr>
<tr>
<td>IV</td>
<td>IV</td>
</tr>
</tbody>
</table>

- **REAL** Determines the real part of certain types
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>CD</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **IMAG** Determines the imaginary part of certain types
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>RE</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>CD</td>
<td>DA</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **CMPLX** Converts types to complex
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Argument</td>
<td>Result</td>
</tr>
<tr>
<td>RE</td>
<td>CM</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>DA</td>
<td>CD</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
</tr>
</tbody>
</table>
- **CONJ** Determines the complex conjugate of certain types

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
<tr>
<td>CM</td>
<td>CM</td>
</tr>
<tr>
<td>DA</td>
<td>DA</td>
</tr>
<tr>
<td>CD</td>
<td>CD</td>
</tr>
<tr>
<td>TM</td>
<td>TM</td>
</tr>
</tbody>
</table>

- **INT** Determines the integer part

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **NINT** Determines the nearest integer

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>

- **NOT** Returns the negation of a logical

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO</td>
<td>LO</td>
</tr>
</tbody>
</table>

- **TRIM** Removes the space characters from the end of a string

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
</tbody>
</table>

- **LTRIM**

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>ST</td>
</tr>
</tbody>
</table>

- **GRIU** Returns the internally allocated graphics output unit number

<table>
<thead>
<tr>
<th>Type</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>RE</td>
</tr>
</tbody>
</table>
A.4 Intrinsic Procedures

The following is a list of all available procedures. The arguments and their properties are listed behind each name. For each of the arguments, ‘v’ denotes that it has to be passed as a variable, usually because a value is assigned to it, and a ’c’ denotes that it can either be passed as a constant or a variable, and no value is assigned to it.

- **MEMALL (v)**
  Returns the total amount of COSY memory that is currently allocated.

- **MEMFRE (v)**
  Returns the total amount of COSY memory that is currently still available.

- **MEMDPV (cc)**
  Performs a dump of the memory contents of a variable. Arguments are the output unit number and the variable name.

- **MEMWRT (c)**
  Writes memory to file: I, NBEG, NEND, NMAX, NTYP, CC, NC in first lines, and CC, NC in subsequent ones. Argument is the unit number.

- **SCRLEN (c)**
  Sets the amount of space scratch variables are allocated with. When needed, use this before calling the corresponding procedure or function. When a negative number is given, it returns the current amount.

- **CPUSEC (v)**
  Returns the elapsed CPU time in the process. It may be necessary to adjust the subroutine CPUSEC in dafox.f depending on the local system.

- **PWTIME (v)**
  Returns the elapsed wall-clock time (sec) on the local node in parallel execution. In serial execution, returns the same time as CPUSEC.

- **PNPRO (v)**
  Returns the total number of concurrent processes in parallel execution, which is in most cases equivalent to the total number of processors used to run the parallel COSY program. In serial execution, the number returned is 1.

- **PROOT (v)**
  Returns 1 if the calling process is a root process in parallel execution, and 0 otherwise. In serial execution, the number returned is 1.

- **QUIT (c)**
  Terminates execution; argument = 1 triggers whatever system traceback is available by performing the deliberate illegal operation sqrt(-1.D0).

- **OS (c)**
  Triggers a system call. For example, a Unix/Linux command ‘date’ can be called by “ OS ’date’ ; ”.

- **OPENF (ccc)**
  Opens a file. Arguments are unit number, filename (string), and status (string, using same syntax as the Fortran open).
- **OPENFB** (ccc)
  Opens a binary file. Arguments are unit number, filename (string), and status (string, same as in Fortran open).

- **CLOSEF** (c)
  Closes a file. Argument is the unit number.

- **NEWF** (c)
  Rewinds a file. Argument is the unit number.

- **BACKF** (c)
  Backspaces a file. Argument is the unit number.

- **READB** (cv)
  Reads a variable in binary form. The arguments are the unit number and the variable name.

- **WRITEB** (cc)
  Writes a variable in binary form. The arguments are the unit number and the variable name.

- **ARIS** (c)
  Sets the arithmetic error handling switch for certain verified types. If the argument is 1, the computation for IN, IV, and TM is carried on even when domain violations, excess remainder bounds (see TMTOL), or other runtime errors happen. In such a case, the error flag which can be retrieved by ARI is set. By default, the value of the switch is 1.

- **DAINI** (ccccv)
  Initializes the order and number of variables of DA, CD or TM. Arguments are order, number of variables, output unit number (nonzero value will trigger output of internally used addressing arrays to the given unit), and the number of resulting monomials (on return).

- **DANOT** (c)
  Sets momentary truncation order for DA, CD and TM.

- **DANOTW** (cc)
  Sets weighted order factor of each independent variable for DA, CD and TM. Arguments are the array containing the weight factors and the size of the array. Must be called before DAINI if needed; incorrect use of DANOTW may void the entire DA, CD and TM computations. Consult us if it is necessary to use this procedure.

- **DAEPS** (c)
  Sets garbage collection tolerance, also called cutoff threshold, for coefficients of DA, CD and TM vectors.

- **DAEPSM** (v)
  Returns the garbage collection tolerance, also called cutoff threshold, for coefficients of DA, CD and TM vectors.

- **EPSMIN** (v)
  Returns the underflow threshold, the smallest positive number representable on the system. This number is determined automatically whenever DA, CD, IN, IV, or TM are used.

- **DAFSET** (c)
  Sets the DA filtering mode. Provide a template DA vector for filtering operations DAFILT and some others including DA multiplications for DA, CD and TM. If the argument is 0 or DAINI is called, the filtering mode is turned off.
A.4 Intrinsic Procedures

- **DAFILT** (cv)
  Filters a DA or CD vector through the template DA vector specified by DAFSET. Arguments are the incoming and the result DA or CD vectors.

- **DAPEW** (cccc)
  Prints the part of DA vector that has a certain order \( n \) in a specified independent variable \( x_i \). Arguments are the unit number, the DA vector, the independent variable number \( i \), and the order \( n \).

- **DAREA** (cvc)
  Reads a DA vector. Arguments are the unit number, the variable name and the number of independent variables.

- **DAPRV** (cccc)
  Writes an array of DA vectors. Arguments are the array, the number of components, maximum and current main variable number, and the unit number.

- **DAREV** (vcccc)
  Reads an array of DA vectors. Arguments are the array, the number of components (limited to 5 currently), maximum and current main variable number, and the unit number.

- **DAFLO** (ccvc)
  Computes the DA representation of the flow of \( x' = f(x) \) for time step 1 to nearly machine accuracy. Arguments: array of right hand sides, the initial condition, result, and dimension of \( f \).

- **CDFLO** (ccvc)
  Same as DAFLO but with complex arguments.

- **DAGMD** (ccvc)
  Computes \( \nabla g \cdot f \) Arguments: \( g \) as a DA, \( f \) as an array of DA, the result DA, and the dimension of \( f \).

- **RERAN** (v)
  Returns a random number between \(-1\) and 1.

- **DARAN** (vc)
  Fills a DA vector with random entries between \(-1\) and 1. Arguments are DA vector and the sparsity fill factor, i.e. the fraction of the coefficients that will actually be set nonzero.

- **DADIU** (ccv)
  Performs a division by a DA independent variable \( x_i \) if possible. Arguments are the number of the independent variable \( i \), and the incoming and the result DA or CD vectors. If the division is not possible, 0 is returned.

- **DADMU** (cccv)
  Performs a division then a multiplication by a DA independent variable \( x_i \) (division) if possible, then by \( x_j \) (multiplication). Arguments are the numbers of the independent variables \( i, j \), and the incoming and the result DA or CD vectors. If the division is not possible, 0 is returned.

- **DADER** (ccv)
  Performs the derivation operation on a DA or CD vector. Arguments are the number with respect to which to differentiate and the incoming and the resulting DA or CD vectors.

- **DAINT** (ccv)
  Performs an integration of a DA vector. Arguments are the number with respect to which to integrate and the incoming and the result DA or CD vectors.
- **DAPLU** (cccv)
  Replaces power of independent variable $x_i$ by constant $C$. Arguments are the DA or CD vector, $i$, $C$, and the resulting DA or CD vector.

- **DASCL** (cccv)
  Scales the $i$-th independent variable $x_i$ by the factor $a$. Arguments are the DA, $i$, $a$, and the resulting DA.

- **DATRN** (cccccv)
  Transforms independent variables $x_i$ with $a_i x_i + c_i$ for $i = m_1, \ldots, m_2$. Arguments are the DA, $a_i$ and $c_i$ supplied by arrays, $m_1$, $m_2$, and the resulting DA.

- **DASGN** (ccvv)
  Flips signs of coefficients of a DA vector by flipping the signs of independent variables to make the first $N_s$ linear coefficients positive. Arguments are the DA, $N_s$, then the array containing the signs of original linear coefficients with the size at least $N_s$, and the resulting DA are returned.

- **DAPEE** (ccv)
  Returns a coefficient of a DA or CD vector or a Taylor Model. Arguments are the DA or CD vector or Taylor Model, the id for the coefficient in TRANSPORT notation (for example, the id for the $x_1 x_2^2$ term is 133), and the returning real or complex number.

- **DAPEA** (cccv)
  Same as DAPEE, except the coefficient is specified by an array with each element denoting the exponent. The third argument is the size of the array.

- **DANORO** (cccvv)
  Computes the norms of power sorted parts of the DA. The power sorting is performed with respect to the $i$-th variable $x_i$. Arguments are the DA, $i$, the size of the array (the next argument), then the norms stored in the array, and the maximum power $n_i$ of $x_i$ existing in the DA are returned. The maximum norms are computed for $\tilde{c}$, and $\tilde{c}(k + 1)$ represents the norm of the $k$-th power part of the DA. The number of returned elements of $\tilde{c}$ is $n_i + 1$. If 0 is given for $i$, an order sorting is performed. For weighted order DA computation, $n_i$ and $k$ denote the weight divided power.

- **DANORS** (cccvv)
  Computes the summation norms of power sorted parts of the DA. The feature is the same with DANORO except that DANORO computes maximum norms.

- **DACLIW** (ccv)
  Extracts “linear” coefficients of a DA or TM. When order weighted DA is used, it extracts order weighted coefficients. Arguments are the DA or TM, the size of the array (the next argument), and the array containing “linear” coefficients.

- **DACQLC** (ccvv)
  Extracts coefficients up to second order of a DA or TM. When order weighted DA is used, it extracts order weighted coefficients. Arguments are the DA or TM, and the size of arrays to store the Hessian matrix and “linear” coefficients. The returning arguments are the two dimensional array for the Hessian matrix $H$, the one dimensional array for the “linear” coefficients $L$, and a real number for the constant $c$. The quadratic part has the form $x^t H x/2 + L x + c$.

- **DAPEP** (cccv)
  Returns a parameter dependent component of a DA or CD vector. Arguments are the DA or CD vector, the coefficient id in TRANSPORT notation for the first $m$ variables, $m$, and the resulting DA or CD vector. The order of resulting DA or CD is lowered by the amount indicated by id.
A.4  Intrinsic Procedures

- **DANOW** (ccv)
  Computes the order weighted norm of the DA vector in the first argument. The other arguments are the weight and the result.

- **DAEST** (cccv)
  Estimates the size of \( j \)-th order terms of the DA vector (with respect to the \( i \)-th variable \( x_i \) if \( i > 0 \)). Arguments are the DA, \( i \), and \( j \), then the estimated size as summation norm is returned.

- **MTREE** (vvvvvv)
  Computes the tree representation of a DA array. Arguments: DA array, elements, coefficient array, 2 steering arrays, elements, length of tree.

- **CDF2** (vvvvv)
  Lets \( \exp(\mathbf{f}_2) \) act on first argument in Floquet variables. Other Arguments: 3 tunes (2\( \pi \)), result.

- **CDNF** (vvvvvvv)
  Lets \( 1/(1-\exp(\mathbf{f}_2)) \) act on first argument in Floquet variables. Other Arguments: 3 tunes (2\( \pi \)), array of resonances with dimensions, result.

- **CDNFDA** (vvvvvvv)
  Lets \( C_j^\pm \) act on the first argument. Other Arguments: moduli, arguments, coordinate number, total number, epsilon, and result.

- **CDNFD$$ (vvvvv)
  Lets \( S_j^\pm \) act on the first argument. Other Arguments: moduli, arguments, spin argument, total number, epsilon, and result.

- **LINV** (cvcv)
  Inverts a quadratic matrix. Arguments are the matrix, the inverse, the number of actual entries, the allocation dimension, and an error flag (0: no error, 132: determinant is zero or very close to zero).

- **LDET** (cccv)
  Computes the determinant of a matrix. Arguments are the matrix, the number of actual entries, the allocation dimension, and the determinant.

- **LEV** (cvvvec)
  Computes the eigenvalues and eigenvectors of a matrix. Arguments are the matrix \( A \), the real and imaginary parts of eigenvalues, a matrix \( V \) containing eigen vectors as column vectors, the number of actual entries, and the allocation dimension. If the \( i \)-th eigenvalue is complex with positive imaginary part, the \( i \)-th and \((i+1)\)-th columns of \( V \) contain the real and imaginary parts of its eigenvector.

- **MBLOCK** (cvvcc)
  Transforms a quadratic matrix to a blocks on diagonal. Arguments are matrix, the transformation matrix and its inverse, allocation and actual dimension.

- **LSLINE** (cccvv)
  Computes the least square fit line \( y = ax + b \) for \( n \) pairs of values \((x(i), y(i))\). Arguments are the array \( x() \), \( y() \), and the number of pairs \( n \), then \( a \) and \( b \) are returned.

- **SUBSTR** (cccv)
  Returns a substring. Arguments are string, first and last numbers identifying substring, and substring.

- **STCRE** (cv)
  Converts a string to a real. Argument are the string and the real.
• **RE CST** ( **ccv** )  
  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.

• **VELSET** ( **vcc** )  
  Sets a component of a vector of reals VE. Arguments are the vector, the number of the component, and the real value for the component to be set.

• **VELGET** ( **ccv** )  
  Returns a component of a vector of reals VE. Arguments are the vector, the number of the component, and on return the real value of the component.

• **VEDOT** ( **ccv** )  
  Computes the scalar (inner, dot) product of vectors. Arguments are the two vectors VEs, and on return the scalar product.

• **VEUNIT** ( **cv** )  
  Normalizes the vector. Arguments are the vector VE to be normalized, and on return the normalized unit vector VE.

• **VEZERO** ( **vvv** )  
  Sets any components of vectors in an array to zero if the component exceeds a threshold value. Arguments are the array of real vectors VE, the number of VE array elements to be checked, and the threshold value. VEZERO is used in repetitive tracking to prevent overflow due to lost particle.

• **VEFILL** ( **vcccc** )  
  Fills a vector array with points on a regular grid and additional random points. Arguments: VE array, domain (IN or IV), dimension, number of points per dimension, number of additional random points.

• **IMUNIT** ( **v** )  
  Returns the imaginary unit \( i \).

• **LTRUE** ( **v** )  
  Returns the logical value true.

• **LFALSE** ( **v** )  
  Returns the logical value false.

• **INTERV** ( **ccv** )  
  Produces an interval from 2 numbers. Arguments are the lower and upper bounds and the resulting interval.

• **INSRND** ( **c** )  
  Enables (1) and disables (0) outward rounding of intervals. By default the rounding is enabled. Extra caution has to be used when disabling outward rounding, because it will void verified computations.

• **INSRF** ( **c** )  
  Sets the factor \( f \). \( f \cdot \varepsilon \) is the outward rounding constant for interval intrinsic functions, where \( \varepsilon \) 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.
• **INTSEC (ccvv)**
  Takes the intersection of the two intervals. Arguments are the two intervals, and the intersection interval and the error code are returned. If there is no intersection between the two intervals, [0,0] is returned with the error code 1. If there is the intersection, the error code is 0. If interval vectors are supplied instead of intervals, the task is performed component-wise.

• **INTUNI (ccvv)**
  Takes the union of the two intervals. Arguments are the two intervals, and the union interval and the error code are returned. The error code is 0 if there is the intersection between the two intervals, and 1 if there is no intersection. If interval vectors are supplied instead of intervals, the task is performed component-wise.

• **INTINC (ccv)**
  Checks if the point or interval (the 2nd argument, RE or IN) is included in the interval (the 1st argument, IN). The error code is returned with 0 if included, and 1 if not. If interval vectors are supplied instead of intervals, the task is performed component-wise. In this case, the 1st argument is IV, the 2nd argument is VE or IV, and the error code is VE.

• **IVSET (v cc)**
  Sets a component of an interval vector IV. Arguments are the interval vector, the number of the component, and the interval for the component to be set.

• **IVGET (ccv)**
  Returns a component of an interval vector IV. Arguments are the interval vector, the number of the component, and on return the interval of the component.

• **INTPOL (v c)**
  Determines coefficients of Polynomial satisfying $P(±1) = ±1, P^{(i)}(±1) = 0, i = 1, ..., n$. Arguments: coefficient array, n.

• **TMVAR (ccv)**
  Creates the $i$-th identity Taylor Model. Arguments: $i$, the Taylor Model domain specification $m_T$, and the resulting $i$-th identity Taylor Model (TM). If $m_T = 1$, the domain is $[-1, 1]$ for all the dimensions. If $m_T = 0$, the domain is $[0, 1]$ for all the dimensions. If a mixture of $[-1, 1]$ and $[0, 1]$ is desired for the domain, specify it for each dimension using the COSY vector (VE) notation. For example, if the domain $[-1, 1] \times [0, 1] \times [-1, 1]$ is desired for a three dimensional system, specify $m_T$ as 1&0&1.

• **TMAVAR (c cccv)**
  Creates a Taylor Model by assembling the components. Arguments: a DA for the polynomial part, the remainder bound interval (IN), the Taylor Model domain specification $m_T$, and the resulting Taylor Model (TM). See TMVAR about $m_T$.

• **TMTOL (c)**
  Sets overflow threshold for the width of remainder bounds of Taylor Models. If the value is exceeded, the Taylor Model is set to be invalid, and the error flag that can be retrieved by ARI is set.

• **TMTOLR (v)**
  Returns the current overflow threshold for remainder bound intervals of TMs.

• **TMNOL (ccv)**
  Truncates a Taylor Model (TM) to a lower order and absorbs discarded orders to remainder bound. Arguments: TM to be truncated, truncation order, resulting TM.

• **TMREA (ev)**
  Reads a Taylor Model (TM). Arguments are the unit number and the variable name.
- **TMREAB** (cv)
  Reads a Taylor Model (TM) in binary form. Arguments are the unit number and the variable name. This procedure will not be available in the future as it can be replaced by READB.

- **TMWRTB** (cc)
  Writes a Taylor Model (TM) in binary form. Arguments are the unit number and the variable name. This procedure will not be available in the future as it can be replaced by WRITEB.

- **DAEXT** (cv)
  Extracts the polynomial part (DA) from a Taylor Model (TM). Arguments are the Taylor Model (TM) and the resulting extracted DA.

- **TMRBND** (cv)
  Extracts the remainder bound interval from a Taylor Model (TM). Arguments are the Taylor Model (TM) and the resulting remainder bound interval.

- **TMDOM** (cv)
  Extracts the domain intervals from a Taylor Model (TM). Arguments are the Taylor Model (TM) and the domain interval (IN) or intervals (IV).

- **LDBL** (ccvv)
  Bounds the minimum of a Taylor Model with the LDB algorithm. Arguments are the Taylor Model (TM) and the tolerance demand. Then, the number of LDB iterations, the reduced region $D_r$ containing the minimizer (IN or IV) and the interval enclosure of the minimum (IN) are returned.

- **LDBC** (ccvvv)
  Checks if a Taylor Model is above the cut-off threshold value $C$, using the LDB algorithm. Arguments are the Taylor Model (TM), the tolerance demand, and the initial cut-off threshold value $C$. Returned are the result flag LC, the number of LDB iterations, and the reduced region $D_r$ (IN or IV). If the Taylor Model is above $C$, LC = 0 is returned. In this case, the returned domain $D_r$ is irrelevant. If it cannot be verified that the Taylor Model is above the cut-off threshold value, LC = 1 is returned and $C$ is updated with an upper bound of the minimum of the Taylor Model, and the reduced region $D_r$ will contain the domain of the Taylor Model below $C$.

- **QMLOC** (ccvvvv)
  Locates a minimizer of the quadratic part of the DA inside the domain [-1,1]. If the quadratic part is positive definite, an lower bound of minimum is estimated using the QFB algorithm. The arguments are the DA or TM, $\epsilon$, and the cut-off threshold value $C$ to be given. For the given limit number of iteration, the actual iteration number $N$ is returned using the same argument. Then, the diagnostic code LQ is returned. And, for the given $x_m$ to start the minimizer search, a minimizer candidate $x_m$ after $N$ iteration is returned using the same argument. And, the lower bound of the minimum estimated with the QFB algorithm is returned.

- **TMINT** (ccv)
  Performs an integration of a Taylor Model (TM). Arguments are the number with respect to which to integrate, the Taylor Model (TM) to be integrated and the resulting Taylor Model (TM).

- **TMPLU** (ccccv)
  Replaces power of independent variable $x_i$ by constant $C$ with validation. Arguments are the Taylor Model (TM), $i$, $C$, id for $C$, and the resulting Taylor Model (TM). id = 0 for $C = 0$, id = 1 for $C = 1$, id = -1 for $C = -1$, and id = 2 for any other $C$.

- **TMTRN** (cccccv)
  Transforms independent variables $x_i$ to $a_i x_i + c_i$ for $i = m_1, \ldots, m_2$ with verification. Arguments are the Taylor Model (TM), $a_i$ and $c_i$ supplied by arrays, $m_1$, $m_2$, and the resulting Taylor model (TM).
• **TMPRIS (c)**
  Sets the Taylor Model (TM) output option parameter. If the argument is 1, the order bound intervals are output as well. By default, the value of the parameter is 0.

• **CLEAR (v)**
  Clears a graphics object.

• **GRMOVE (cccv)**
  Appends one move to a graphics object. Arguments are the three coordinates and the graphics object.

• **GRDRAW (cccv)**
  Appends one draw to a graphics object. Arguments are the three coordinates and the graphics object.

• **GRCHAR (cv)**
  Adds a string at the current position in a graphics object. Arguments are the string and the graphics object.

• **GRCOLR (cv)**
  Adds a color change to a graphics object. Arguments are the new color id and the graphics object.

• **GRWDTH (cv)**
  Adds a width change to a graphics object. Arguments are the new width id and the graphics object.

• **GRDOT (cccv)**
  Appends one move and one dot to a graphics object. Arguments are the three coordinates and the graphics object.

• **GRCURV (cccccccv)**
  Appends one B-Spline to a graphics object. 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.

• **GRPROJ (cv)**
  Sets the 3D projection angles of a graphics object. Arguments are phi and theta in degrees and the graphics object.

• **GRZOOM (cccccvc)**
  Sets the 3D zooming area specified by two points \((x_1, y_1, z_1)\) and \((x_2, y_2, z_2)\) of a graphics object. Arguments are \(x_1, x_2, y_1, y_2, z_1, z_2\), and the graphics object.

• **GRMIMA (cvvvvvv)**
  Finds the minimal and the maximal coordinates in a graphics object. Arguments are the object and \(x_1, x_2, y_1, y_2, z_1, z_2\).

• **RKCO (vvvvv)**
  Sets the coefficient arrays used in the COSY eighth order Runge Kutta integrator.

• **POLSET (c)**
  Sets the polynomial evaluation method used in POLVAL. 0: expanded, 1: Horner.

• **POLVAL (cccccvc)**
  Performs the POLVAL composition operation. See section 2.6 for details.
Index

:=, 19
; , 19
<, 60
>, 60
| , 61
*, 58
+, 56
-, 57
/, 59
=, 60
#, 60
%, 61
&, 14, 24, 60
ˆ, 59

ABS (Intrinsic Function), 71
Absolute Value, 71
Acknowledgements, 52
ACOS (Intrinsic Function), 68
AD, see Automatic Differentiation
Addition (Operator), 56
Algorithm (Optimization), 22
Allocation Size
   Actual of Object, 66
   For all Types, 64
AND, 58
Anti-Derivation, 77
   Example of Use, 14
Anti-Derivation (Operator), 61
AquaTerm Graphics, 8, 45
ARI (Intrinsic Function), 67
ARI (Intrinsic Function), 76, 81
ARIS (Intrinsic Procedure), 76
Array
   C++, 31, 34
   COSYScript, 19
   F90, 41
ASCII low resolution Graphics, 44
ASIN (Intrinsic Function), 68
Assignment, 19, 20
Assignment (COSYScript command), 18
ATAN (Intrinsic Function), 69
Automatic Differentiation, 5, 6
Axes for Graphics, 44
B Spline (in Graphics), 83
BACKF (Intrinsic Procedure), 76
Backspace File, 76
Backwards Compatibility, 10
BEGIN (COSYScript command), 18, 19
Binary
   Open File, 76
   READ, 23, 76
   WRITE, 23, 76
Bounder
   LDB, 82
   LDB, Use for Cutoff, 82
   QFB, 82
C++, 6
   Array Access, 31
   Arrays, 34
   Assignment Operators, 30
   Constructors, 29
   COSY Procedures, 33
   Elementary Operations, 32
   Interface, 28
   Intrinsic Functions, 32
   Memory Management, 29
   Operator new, 29
   Printing, 31
   Streams, 31
   Type Conversion, 32
   Unary Operators, 31
CALL SYSTEM, 75
CD, 14
CD (Intrinsic Function), 63
CD (COSY Object), 55
CDF2 (Intrinsic Procedure), 79
CDFLO (Intrinsic Procedure), 77
CDNF (Intrinsic Procedure), 79
CDNFDA (Intrinsic Procedure), 79
CDNFDS (Intrinsic Procedure), 79
CG (Curve Graphics), 44
Checkpointing, 6
Class Cosy, 28
CLEAR (Intrinsic Procedure), 83
Close File, 23, 76
CLOSEF (Intrinsic Procedure), 76
Cluster Environments, see MPI
CM, see Complex
CM (Intrinsic Function), 62
CM (COSY Object), 55
CM (Intrinsic Function)
   Example of Use, 12
CMPLX (Intrinsic Function), 73
Coefficient, 61
   Flushing, 76
INDEX

of Complex DA Vector - Get, 61
of DA Vector - Get, 61, 78
of Taylor Model, 61
Color (Graphics), 83
Command Line, 9
Compaq Fortran, 9
Compatibility
  Compilers Tested with COSY, 9
  Earlier Versions, 10
Complex, 12
  Conversion from Others, 62, 73
  Conversion to String, 62
  Float, 73
  Imaginary Part, 61, 73
  Imaginary Unit, 80
  Output Format, 23
  Real Part, 61, 73
Complex DA Vector
  Coefficient - Get, 61
  Conversion from Others, 63, 73
  Output Format, 24
Component, 61
  of Interval Vector - Get, 81
  of Complex DA Vector - Get, 61
  of DA Vector - Get, 61, 78
  of Interval Vector - Get, 61
  of Interval Vector - Set, 81
  of Taylor Model - Get, 61
  of Vector - Get, 61, 80
  of Vector - Set, 80
Composition (COSY Operator)
  Use in C++, 33
Composition (Operator), 60
  Example of Use, 13, 14
Computational Differentiation, see Automatic Differentiation
Concatenation (Operator), 60
CONJ (Intrinsic Function), 74
CONS (Intrinsic Function), 72
Constant Part, 72
Control, 5
Control Statements, 21
COS (Intrinsic Function), 68
COSH (Intrinsic Function), 69
COSY
  Installation, 7
  Language (COSYSript), 18
  Obtaining Source, 6
  References, 6
Cosy class, 28
COSY Language, see COSYSript
COSY-GO, 6
COSY-VI, 6
COSY fox, 6
COSY.ARRAYGET (F90 Subroutine), 36
COSY.ARRAYSET (F90 Subroutine), 36
COSY.CREATE (F90 Subroutine), 36
COSY.DESTROY (F90 Subroutine), 36
COSY.DOUBLE (F90 Subroutine), 36
COSY.GETTEMP (F90 Subroutine), 36
COSY.INIT (F90 Subroutine), 35
COSY.LOGICAL (F90 Subroutine), 36
COSY.TMP (F90 Subroutine), 36
COSY.WRITE (F90 Subroutine), 36
COSYSript, 5
  Arrays, 19
  Assignment, 20
  BEGIN, 19
  ELSEIF, 21
  END, 19
  ENDFIT, 22
  ENDFUNCTION, 19
  ENDFIT, 21
  ENDOCLASS, 22
  ENDPLOO, 18, 25
  ENDPROEDURE, 19
  ENDTREWHILE, 21
  FIT, 22
  Flow Control, 21
  FUNCTION, 19
  Function call, 20
  GUIIO, 48
  IF, 21
  INCLUDE, 24
  Locality and Globality, 20
  LOOP, 21
  PLOO, 18, 25
  PNPRO, 26
  PROCEDURE, 19
  Procedure call, 20
  PROOT, 26
  PWTIME, 26
  VARIABLE, 19
  WHILE, 21
COSYSript, 18
CPU Time, 9, 75
CPUSEC (Intrinsic Procedure), 9, 75
Curve, Drawing of, 44
Cutoff Threshold, 76
Cygwin Installation, 9
DA, see DA Vector
DA (Intrinsic Function), 63
DA (COSY Object), 55
DA (Intrinsic Function)
  Example of Use, 14
DA Filtering, 76, 77
DA Runge Kutta, 83
DA Vector, 5, 14
  Array Output (DAPRV), 77
  Coefficient - Get, 61, 78
  Constant Part, 62
  Conversion from Others, 63
  Derivation, 77
  Estimate Size, 79
  Extract from Taylor Model, 82
  Extracting Linear Coefficients, 78
  Extracting Quadratic Form, 78
  Filtered Output (DAPEW), 77
  Filtering, 76, 77
  Get Coefficient, 78
  Initialization, 76
  Norm, 79
  Output Format, 23
  Parameter Dependent Coefficient, 78
  Plugging in one Component, 78
  Read (DAREA), 77
  Remainder, see Taylor Model
  Scaling, 78
  Shift and Scale, 78
  Sign Normalization, 78
  Truncation Order, 76
DACLIW (Intrinsic Procedure), 78
DACQLC (Intrinsic Procedure), 78
DADER (Intrinsic Procedure), 77
DADIU (Intrinsic Procedure), 77
DADMU (Intrinsic Procedure), 77
DAE, 6
DAEPS (Intrinsic Procedure), 76
DAEPSM (Intrinsic Procedure), 76
DAEST (Intrinsic Procedure), 79
DAEXT (Intrinsic Procedure), 82
DAFILT (Intrinsic Procedure), 77
DAFLO (Intrinsic Procedure), 77
dafx, 9
DAFSET (Intrinsic Procedure), 76
DAGMD (Intrinsic Procedure), 77
DAINI (Intrinsic Procedure), 76
DAINI (Intrinsic Procedure)
  Example of Use, 14
DAINT (Intrinsic Procedure), 77
DANORO (Intrinsic Procedure), 78
DANORS (Intrinsic Procedure), 78
DANOT (Intrinsic Procedure), 76
DANOTW (Intrinsic Procedure), 76
DANOW (Intrinsic Procedure), 79
DAPEA (Intrinsic Procedure), 78
DAPEE (Intrinsic Procedure), 78
DAPEP (Intrinsic Procedure), 78
DAPEW (Intrinsic Procedure), 77
DAPEX (Intrinsic Procedure), 78
DAPEW (Intrinsic Procedure), 77
DAPRV (Intrinsic Procedure), 77
DAPRV (Intrinsic Procedure), 74
DARAN (Intrinsic Procedure), 77
DAREA (Intrinsic Procedure), 77
DAREV (Intrinsic Procedure), 77
DASCL (Intrinsic Procedure), 78
DASGN (Intrinsic Procedure), 78
DATRN (Intrinsic Procedure), 78
Debug
  Memory Dump, 75
  Memory of Variable, 75
DEC Fortran, 9
Declaration, 19
Derivation, 61, 77
Derivation (COSY Operator)
  Use in C++, 33
  Example of Use, 15
Derivative, 61, 77
Determinant, 79
Differential Algebra, 5
  Derivation, 77
  Flow Computation, 77
  Structure for Taylor Models, 82
Differential Algebra Remainder, see Taylor Model
Differentiation, 61
Division (Operator), 59
Domain Error
  Veriﬁed Types, 67
Domain of Taylor Model
  Get, 82
  Set, 81
Dot (Graphics), 83
Draw (Graphics), 83
Dual Core, 7, 8
Dump
  Entire Memory, 75
  Variable, 75
Dynamic Typing, 6, 18

  Earlier Versions, Compatibility, 10
  Eigenvalues and Eigenvectors, 79
  Elapsed Time, 75
  Elementary C++ Operations, 32
ELSE (equivalent COSYSscript command), 21

ELSE (equivalent COSYSscript command), 21
ELSEIF (COSYScript command), 18, 21
END (COSYScript command), 18, 19
ENDIF (COSYScript command), 18, 21
ENDFUNCTION (COSYScript command), 18, 19
ENDIF (COSYScript command), 18, 21
ENDLOOP (COSYScript command), 18, 22
ENDFUNCTION (COSYScript command), 18, 19
ENDPROCEDURE (COSYScrip command), 18, 19
ENDWHILE (COSYScrip command), 18, 21
EPSMIN (Intrinsic Procedure), 76
Equal (Operator), 60
ERF (Intrinsic Function), 71
Error Function
  Complex, 71
  Real, 71
Error Handling, Verified Types, 76
Error Messages, 26
Executable Statements, 20
Execution Termination, 75
EXP (Intrinsic Function), 67
Exponentiation (Operator), 59
Extraction (COSY Operator)
  Use in C++, 33
Extraction (Operator), 61
  Example of Use, 13–15

F2C
  Converter, 28
F90, 6
  .DI., 39
  .EQ., 38
  .EX., 39
  .GT., 38
  .LT., 38
  .LE., 38
  .UN., 39
Addition, 37
Arrays, 41
Assignment, 39
Concatenation, 39
Derivation, 39
Division, 38
Exponentiation, 38
Extraction, 39
Functions, 40
Interface, 35
Memory Management, 40
Multiplication, 37
Operations, 37
Power Operation, 38
Subroutines, 40
Subtraction, 37

Utility Routines, 35
False (Logical), 80
FG (Frame Graphics), 44
File
  Backspace, 76
  Binary, 76
  Close, 76
  Open, 75
  Rewind, 76
File Handling, 9
Filtering, 76, 77
FIT (COSYScrip command), 18, 22
Fitting, 22
Float to Complex, 73
Flow, 6
Flow Control Statements, 21
Flow, DA representation, 77
Flushing of Coefficients, 76
Format, 80
  Default Output, 23
  Output, 24
Formatted
  Input, 23
  Output, 23
Fortran Output Format, 24
foxfit, 9
foxgraf, 9, 10
foxy, 9
Frame for Graphics, 44
Function
  Call in COSYScrip, 20
  Drawing of, 44
  F90, 40
  Intrinsics, 62
  Local, 20
FUNCTION (COSYScrip command), 18, 19
Garbage Collection Tolerance, 76
GENFOX, 55
genfox.dat, 55
GKS Graphics, 10
GNU Fortran, 9
  Installation, 8
GR, see Graphics, see Graphics
GR (COSY Object), 55
Graphical User Interface, 10, 46
  Activate, 49
  Advanced, 48
  Alignment, 49
  Automatic, 47
  Button, 49
  Center, 49
Close, 49
Commands, 48
Console, 49
Custom Executable, 52
Deactivate, 49
Debug, 49
Examples, 51
Finish, 49
Focus, 49
GUIIO, 48
Image, 49
Java GUI, 52
Just, 49
Justified, 49
Layout, 51
Left, 49
Line, 49
NewCell, 49
NewLine, 49
ReadCheckbox, 49
ReadField, 49
Reading, 47, 48
ReadList, 49
ReadNumber, 49
ReadOption, 49
ReadProgress, 49
Reference, 48
Right, 49
Set, 49
Show, 49
Simple, 47
Spacer, 49
Text, 49
Title, 49
Graphics, 5, 9, 44
Adding New Driver, 45
AquaTerm, 45
ASCII (Low Resolution), 44
Axes and Frames, 44
B Spline, 83
Clear, 83
Color, 83
Dot, 83
Draw, 83
GrWin, 9, 45
Interactive, 44
LaTeX, 44
Low-Resolution ASCII, 44
Merging, 44
Meta File, 44, 46
Minimum and Maximum, 83
Move, 83
Output, 24
PGPLOT, 9, 44, 45
PostScript (direct), 44
Projection, 83
Required Low-Level Routines, 45
String, 83
Supported Drivers, 44
Width, 83
Windows, 44
Zooming, 83
Graphical Interface, see Graphical User Interface
GUIIO (COSYScript command), 47, 48
Hessian, 78
High Performance Computing, see MPI
Hyperthreading, 5
Identity
CD Vector, 63
DA Vector, 63
TM, 63
IF (COSYScript command), 18, 21
Illegal Operation SQRT(-1.D0), 27
IMAG (Intrinsic Function), 73
Imaginary Part, 61, 73
Imaginary Unit, 80
IMUNIT (Intrinsic Procedure), 80
IN, see Interval, see Interval
IN (Intrinsic Function), 63
IN (COSY Object), 55
IN (Intrinsic Function)
Example of, 13
Example of Use, 14
INCLUDE (COSYScript command), 18, 24
INL (Intrinsic Function), 72
INDEX

INL (Intrinsic Function), 10
Input, see Read
INSRF (Intrinsic Procedure), 80
INSRND (Intrinsic Procedure), 13, 80
Installation, 7
  Cygwin, 9
  g77, 8
  Linux, 8
  Mac OS X, 7
  New Graphics Packages, 45
  New Optimizer, 43
UNIX, 9
INT (Intrinsic Function), 74
Integer Part, 74
Integral, 61
Integration, 61
Intel, 7, 8
Intel Fortran, 7, 9
Interface
  C++, 28
  F90, 35
INTERV (Intrinsic Procedure), 80
Interval, 13
  Arithmetic Error, 67
  Box Volume, 73
  Containment of Point, 81
  Conversion from Others, 63
  Conversion to String, 62, 80
  Domain of Taylor Model, 82
  Error handling, 76
  Intersection, 81
  Lower Bound, 61, 72
  Midpoint, 62
  Output Format, 24
  Remainder Bound of Taylor Model, 82
  Setting from Bounds, 80
  Union, 81
  Upper Bound, 61, 72
  Verification, 80
  Width, 72
Interval Vector
  Box Volume, 73
  Conversion from Others, 63
  Get Component, 61, 81
  Lower Bound, 72
  Set Component, 81
  Upper Bound, 72
  Width, 72
INTINC (Intrinsic Procedure), 81
INTPOL (Intrinsic Procedure), 81
Intrinsic
  C++ Functions, 33
  F90 Functions, 40
  F90 Subroutines, 40
  Functions, 62
  Objects, 55
  Operators, 56
  Procedures, 75
  Types, 55
INTSEC (Intrinsic Procedure), 81
INTUNI (Intrinsic Procedure), 81
INU (Intrinsic Function), 72
Inverse Matrix, 79
ISRT (Intrinsic Function), 70
ISRT3 (Intrinsic Function), 70
Iterations (Optimization), 22
IV, see Interval Vector
IV (Intrinsic Function), 63
IV (COSY Object), 55
IV (Intrinsic Function)
  Example of Use, 14, 15
IVGET (Intrinsic Procedure), 81
IVOLUM (Intrinsic Function), 73
IVSET (Intrinsic Procedure), 81
IWIDTH (Intrinsic Function), 72

Keywords
  COSYScript List of, 18
  COSYScript Syntax of, 19

LaTeX
  Encapsulated PostScript (EPS) file inclusion, 44
  Graphics, 44
  PostScript (PS) file inclusion, 44
LCD (Intrinsic Function), 65
LCM (Intrinsic Function), 64
LDA (Intrinsic Function), 65
LDB (Intrinsic Function), 72
LDB (Linear Dominated Bounder), 82
  Use for Cutoff, 82
LDBC (Intrinsic Procedure), 82
LDBL (Intrinsic Procedure), 82
LDETH (Intrinsic Procedure), 79
Least square fit, 79
Length, see Allocation Size
LENGTH (Intrinsic Function), 66
Less Than (Operator), 60
LEV (Intrinsic Procedure), 79
Levi-Civita, 6
LFALSE (Intrinsic Procedure), 80
LGR (Intrinsic Function), 65
License, 6
Lie Derivative, 77
Lie Operator, 79
LIN (Intrinsic Function), 64
Line breaks, 19
Line length, 10
Linear Coefficients, 78
Linking Code, 24
Linux, 9
   Installation, 8
LINV (Intrinsic Procedure), 79
LIV (Intrinsic Function), 64
LLO (Intrinsic Function), 64
LMDIF (Optimizer), 42
LMEM, 10
LO, see Logical, see Logical
LO (Intrinsic Function), 62
LO (COSY Object), 55
LO (Intrinsic Function)
   Example of Use, 13
Local Procedures and Functions, 20
LOG (Intrinsic Function), 67
Logical, 5, 13
   AND, 58
   False, 80
   Negation, 74
   OR, 56
   Output Format, 24
   True, 80
LOOP (COSYScript command), 18, 21
LRE (Intrinsic Function), 64
LSLINE (Intrinsic Procedure), 79
LST (Intrinsic Function), 64
LTM (Intrinsic Function), 65
LTRIM (Intrinsic Function), 74
LTRUE (Intrinsic Procedure), 80
LVE (Intrinsic Function), 64

Mac OS X, 9
   Installation, 7
Macro Source Files, 9
Makefile, 8
   C++ Interface, 29
Matrix
   Determinant, 79
   Eigenvalues and Eigenvectors, 79
   Inverse, 79
MBLOCK (Intrinsic Procedure), 79
MEMALL (Intrinsic Procedure), 75
MEMDPV (Intrinsic Procedure), 75
MEMFRE (Intrinsic Procedure), 75
Memory, 10
   Allocated, 75
   Allocated by Variable, 66
   Dump All, 75
   Free, 75
   MEMDPV, 75
   Starting Address of Variable, 66
Memory Management, 10
MEMWRT (Intrinsic Procedure), 75
Merging of Pictures, 44
Meta File, 44–46
Microsoft Windows, 9
   Installation, 7
Minimum and Maximum (Graphics), 83
Move (Graphics), 83
MPI, 10, 25
   Compiling, 25
   mpif77, 25
   mpirun, 25, 26
   Number of Processors, 26, 75
   OpenMPI, 25
   Root Process, 26, 75
   Running COSY, 26
   Wall Clock, 26, 75
MS Windows, 7
MTREE (Intrinsic Procedure), 79
Multicore, 5, 25
Multiplication (Operator), 58
Names, COSYSObject Syntax of, 19
Nearest Integer, 74
Negation, 74
NINT (Intrinsic Function), 74
Norm, 79
NORM (Intrinsic Function), 71
NOT (Intrinsic Function), 74
Not Equal (Operator), 60
Object
   Complex Number, 12
   DA Vector, 14
   Interval, 13
   Logical, 13
   RDA, 15
   Real Number, 12
   String, 12
   Taylor Model, 15
   Vector, 13
   Object Oriented, 6
   Objective Functions, 22
   Objects, 12, 55
   ODE, 6
   ODE, Flow Computation, 77
Open File, 23, 75, 76
OPENF (Intrinsic Procedure), 75
OPENFB (Intrinsic Procedure), 76
OpenMP, 5, 25
Operations, see Operators
Operators, 56
Optimization, 5, 6, 18, 22, 42
DA-based, 43
Including New Algorithm, 43
OR, 56
OS (Intrinsic Procedure), 75
Output Format Default, 23
Parallel Environments, see MPI
Parametric Polymorphism, 6
Parentheses, 20
Pathscale Fortran, 10
PGPLOT Graphics, 9, 44, 45
Picture
Drawing of Function, 44
PLOOP (COSYScript command), 18, 25
PM (Print Map), 24
PNPRO (Intrinsic Procedure), 75
PNPRO (Intrinsic Procedure), 26
Pointer of Variable, 66
POLSET (Intrinsic Procedure), 83
POLVAL (Intrinsic Procedure), 15, 83
Polygon, Drawing of, 44
Polymorphism, 6, 18
Polynomial, 15
Polynomial Bounding
LDB, 82
QFB, 82
PostScript Graphics
Direct, 44
via PGPLOT, 44
Problems, 7
Procedure
C++, 33
Call of in COSYScript, 20
Intrinsics, 75
Local, 20
PROCEDURE (COSYScript command), 18, 19
Processor
Number of in MPI, 26, 75
Program Segments, 19
Projection (Graphics), 83
PROOT (Intrinsic Procedure), 75
PROOT (Intrinsic Procedure), 26
Prototyping, 5, 18
PS Graphics
Direct, 44
via PGPLOT, 44
PWTIME (Intrinsic Procedure), 26, 75
QFB (Quadratic Fast Bounder), 82
QMLOC (Intrinsic Procedure), 82 Quadratic Coefficients, 78
Questions, 7
QUIT (Intrinsic Procedure), 75
QUIT (Intrinsic Procedure), 22
R (COSY Function), 23
Random DA Vector, 77
Random Number, 77
RD, see Taylor Model
RDA, 15
RDREAB (Intrinsic Procedure), 23
RDVAR, 10
RDWRTB (Intrinsic Procedure), 23
RE, see Real
RE (Intrinsic Function), 62
RE (COSY Object), 55
Read, 22
Binary, 23, 76
DA Vector, 77
Formatted, 23
Graphics Meta File, 45
Taylor Model, 81
Taylor Model, Binary, 82
Unformatted, 22
READ (COSYScript command), 18, 22
READB (Intrinsic Procedure), 76
READB (Intrinsic Procedure), 23
Real, 5, 12
Conversion from Others, 62
Conversion to Interval, 63
Conversion to String, 62, 80
Output Format, 23
REAL (Intrinsic Function), 73
Real Part, 61, 73
of Complex, 62
of Complex DA Vector, 63
RECST (Intrinsic DA Vector), 80
RECST (Intrinsic Procedure), 24
Remainder Bound
Flushing of Coefficients, 76
Remainder Differential Algebra, see Taylor Model
RERAN (Intrinsic Procedure), 77
Reverse Communication, 43
REWOF (Intrinsic Procedure), 76
Rewind File, 76
RKCO (Intrinsic Procedure), 83
Root Process, 26, 75
Runge Kutta Coefficients, 83
SAVE (COSYScript command), 18, 24
Scaling of DA Vector, 78
Scratch Variables
   Allocation Size, 75
SCRLEN (Intrinsic Procedure), 75
Semicolon, 19
SF (COSY Function), 24
Shift and Scale
   DA Vector, 78
   Taylor Model, 82
Simplex Algorithm, 42
Simulated Annealing, 42
Simulation Prototyping, 5
SIN (Intrinsic Function), 67
SINH (Intrinsic Function), 69
Size, see Allocation Size
Sparsity, 6
Spline (in Graphics), 83
Splitting Code, 24
SQR (Intrinsic Function), 70
SQRT (Intrinsic Function), 70
SQRT(-1.D0), 27
ST, see String
ST (Intrinsic Function), 62
ST (COSY Object), 55
ST (Intrinsic Function)
   Example of Use, 24
   Example of User, 13
STCRE (Intrinsic Procedure), 79
Stop, 75
String, 5, 12
   Conversion from Interval, 80
   Conversion from Others, 62
   Conversion from Real, 80
   Output to Graphics, 83
   Remove, 74
   Substring, 61, 79
String Output, 23
Structuring, 19
Sub-Vector, 61
SUBSTR (Intrinsic Procedure), 79
Substring, 61, 79
   Read to Real, 23
Subtraction (Operator), 57
Support, 7
Syntax Changes, 10
Syntax Table (COSYScript Language), 18
SYSTEM, 75
System Time, 75
System Traceback, 27, 75
TAN (Intrinsic Function), 68
TANH (Intrinsic Function), 69
Taylor Model, 5, 10, 15
   Anti-Derivation, 82
   Arithmetic Error, 67
   Bound by LDB, 82
   Bound by QFB, 82
   Coefficient - Get, 61
   Constant Part, 62
   Conversion from Others, 64
   Cutoff test by LDB, 82
   Domain Set, 81
   Domain Shift and Scale, 82
   Error handling, 76
   Extract Polynomial Part, 82
   Extract Remainder Bound, 82
   Filtering, 76
   Get Domain, 82
   Get Order and Remainder Bounds, 63
   Identity, 81
   Initialization, 76
   Integration, 82
   Order Truncation, 81
   Output Format, 24
   Output Options, 83
   Plug in Variable, 82
   Read, 81
   Read Binary, 82
   Remainder Bound Overflow Threshold, 81
   Set from Components, 81
   Syntax Changes, 10
   Truncation Order, 76
   Verification, 80
   Write Binary, 82
Technical Support, 7
Tektronix, 10
Termination of Execution, 75
Time
   Elapsed, 75
   Wall Clock, 75
TM, see Taylor Model
TM (Intrinsic Function), 64
TM (COSY Object), 55
TM (Intrinsic Function)
   Example of Use, 15
TM.fox, 6
TMAVAR (Intrinsic Procedure), 81
TMDOM (Intrinsic Procedure), 82
TMINT (Intrinsic Procedure), 82
TMNOL (Intrinsic Procedure), 81
TMPLU (Intrinsic Procedure), 82
INDEX

TMPRIS (Intrinsic Procedure), 83
TMRBND (Intrinsic Procedure), 82
TMREA (Intrinsic Procedure), 81
TMREAB (Intrinsic Procedure), 82
TMTOL (Intrinsic Procedure), 81
TMTOL (Intrinsic Procedure), 76
TMTOLR (Intrinsic Procedure), 81
TMTRN (Intrinsic Procedure), 82
TMVAR (Intrinsic Procedure), 81
TMVAR (Intrinsic Procedure), 10, 15
TMWRTB (Intrinsic Procedure), 82
Tolerance (Optimization), 22
Traceback, 75
TRIM (Intrinsic Function), 74
True (Logical), 80
Truncation Order, 76
TYPE (Intrinsic Function), 65
Types, 12, 55
Underflow Threshold, 76
UNIX, 9
   Installation, 9
User’s Agreement, 6

Variable
   Declaration, 19
   Visibility, 20
VARIABLE (COSYSObjective command), 18, 19
VARMEM (Intrinsic Function), 66
VARPOI (Intrinsic Function), 66
VE, see Vector
VE (Intrinsic Function), 63
VE (COSY Object), 55
Vector, 5, 13
   Average, 62
   Concatenate, 60
   Conversion from Others, 63
   Conversion to IV, 63
   Dot Product, 80
   Get Component, 61, 80
   Get Sub-Vector, 61
   Inner Product, 80
   Maximum, 71
   Minimum, 71
   Normalization, 80
   Output, 24
   Scalar Multiplication, 80
   Scalar Product, 80
   Set Component, 80
   Splitting, 61
   Unit Vector, 80
VEDOT (Intrinsic Procedure), 80
VEFILL (Intrinsic Procedure), 80
VELGET (Intrinsic Procedure), 80
VELMAX, see VMAX
VELMIN, see VMIN
VELSET (Intrinsic Procedure), 80
VERSION, 35
Version Compatibility, 10
VEUNIT (Intrinsic Procedure), 80
VEZERO (Intrinsic Procedure), 80
VGA Graphics, 10
VMAX (Intrinsic Function), 71
VMIN (Intrinsic Function), 71
VOLUM (Intrinsic Function), 73
Volume, 73
Wall Clock Time, 75
WERF (Intrinsic Function), 71
WHILE (COSYSObjective command), 18, 21
WIDTH (Intrinsic Function), 72
Width (Graphics), 83
Windows, 9
Write
   Binary, 23, 76
   DA Vector Array, 77
   Memory Dump, 75
   Taylor Model, Binary, 82
WRITE (COSYSObjective command), 18, 23
WRITEB (Intrinsic Procedure), 23, 76
Xcode, 8
XL Fortran, 10
Zooming (Graphics), 83