COSY INFINITY 10.0
Programmer's Manual

MSU Report MSUHEP 151102-rev

M. Berz and K. Makino
Michigan State University

August 2017

Contents

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 INFINITY interested in its inner workings beyond merely using tools based on COSY INFINITY, read the remainder of this chapter.
2. New users of COSY INFINITY, install it on the system of your choice. Follow instructions in Section 1.5 on page 8.
3. COSYScript language users, note the brief syntax table in Section 3.1 on page 29.
   There is a also brief guide to quickly start COSYScript programming in Appendix B on page 90, especially for performing Beam Physics computations.
4. COSYScript 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 Section 7 on page 52.
6. F90 Users, refer to Section 8 on page 59.

1.2 What is COSY INFINITY

COSY INFINITY is an environment for the use of various advanced concepts of modern scientific computing. COSY INFINITY is extensively verified, and currently has more than 2000 registered users. The COSY INFINITY 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 [15] [13] [14] data type. Allows rigorous verified computation under suppression of dependencies. Includes the rigorous treatment of a remainder bound over a given domain. (This data type is not supported in the current version of COSY INFINITY.)
   (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 extensive 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. Libraries for C and F77 of highly optimized common operations for the types.

3. The COSYScript environment to use these data types with the following key features:
   (a) Scripting language that is compiled and executed on the fly, highly optimized for turnaround. No need for linking, and very low interpretative overhead. Geared towards simulation, control, and algorithm prototyping.
   (b) Compactness of syntax and resulting code.
   (c) Object oriented with polymorphism (dynamic typing)
   (d) Local and global optimization (non-verified) built in at the language level
4. A C++ Interface making the types and operations available as a class to be used within C++ user code
5. A F90 Interface to utilize the types and operations as a module to be used within F90 user code

The environment is extensively verified, and currently has more than 2000 registered users. For purposes of maintainability, there is only one source, which is automatically cross-translated to C, and there are tools that automatically generate the C++ and F90 class and module.

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

1. High-order multivariate Automatic Differentiation of functions written C++, F90, and COSYScript 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 INFINITY'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 based on Taylor models with often significantly reduced dependency problem. (Note: Taylor models are not supported in version 10.)
5. COSY-VI [17] [6] [16], a rigorous verified integrator based on approximate differential algebraic flows and Taylor models. (Note: Taylor models are not supported in version 10.)
6. COSY-GO, a rigorous global optimizer based on Taylor models. (Note: Taylor models are not supported in version 10.)
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 [8], and various other applications. For more details, refer to [3]. Note that some of low and high level utility tools can be found in cosy.fox (see the Beam Physics Manual of COSY INFINITY).

1.3 User's Agreement

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

Permitted Uses: Michigan State University ("MSU") grants you, as "End User," the right to use COSY INFINITY for non-commercial purposes only. Registered users will automatically be given access

to updates of the code as they become available. Conversely, we encourage end users to make available tools of sufficient generality they develop for the purpose of inclusion in the master version. Any errors detected in COSY INFINITY should be reported; comments to improve its performance are appreciated. If the code proves useful for work that is being published, a reference is expected.

Prohibited Uses: End User may not make copies of COSY INFINITY available to others, but rather refer them to register for their own license. End User may not distribute, rent, lease, sub-license, decompile, disassemble, or reverse-engineer the COSY INFINITY materials provided by MSU without the prior express written consent of MSU; or remove or obscure the Board of Trustees of MSU copyright notices or those of its licensors. The source files are provided for purposes of compilation only and should not be modified. We advise against modification of the provided COSYScript libraries so as to maintain a clear upgrade path, but rather to maintain derivative code in separate files.

Intellectual Property: COSY INFINITY is a proprietary product of MSU and is protected by copyright laws and international treaty. This Agreement is a legal contract between you, as End User, and the Board of Trustees of MSU governing your use of COSY INFINITY. MSU retains title to COSY INFINITY. You agree to use reasonable efforts to protect the code from unauthorized use, reproduction, distribution, or publication. All rights not specifically granted in this License Agreement are reserved by MSU.

Warranty: MSU MAKES NO WARRANTY, EXPRESS OR IMPLIED, TO END USER OR TO ANY OTHER PERSON OR ENTITY. SPECIFICALLY, MSU MAKES NO WARRANTY OF FITNESS FOR A PARTICULAR PURPOSE OF COSY INFINITY. MSU WILL NOT BE LIABLE FOR SPECIAL, INCIDENTAL, CONSEQUENTIAL, INDIRECT OR OTHER SIMILAR DAMAGES, EVEN IF MSU OR ITS EMPLOYEES HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT WILL MSU LIABILITY FOR ANY DAMAGES TO END USER OR ANY PERSON EVER EXCEED THE FEE PAID FOR THE LICENSE TO USE THE SOFTWARE, REGARDLESS OF THE FORM OF THE CLAIM.

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 INFINITY web site

http://cosyinfinity.org

Installation packages compiled with the Intel Fortran Compiler for Microsoft Windows PC, for Mac OS X, and for Linux are available. The Microsoft Windows PC package is linked with the GrWin graphics library. The Mac OS X package is linked with the AquaTerm graphics library. Please refer to Sections 1.5.1, 1.5.2, and 1.5.3 for the details.

Instead of using the executables in the above installation packages, there may be some situations necessary to install COSY INFINITY by compiling the COSY INFINITY Fortran source files. Typically, such situations happen for Linux systems and UNIX-like systems. The instructions are provided for typical

Linux/UNIX-like installations in Section 1.5.6, and for parallel environments in Section 1.5.8. First please review Section 1.5.4 that describes the COSY INFINITY Fortran source files and Fortran compilers.

The code for COSY INFINITY consists of the macro source files, which depend on the particular application of COSY INFINITY. 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 COSYScript and have to be compiled by the local executable program of COSY INFINITY as part of the installation process. Please see Section 1.7 for running COSY INFINITY to compile COSYScript macro source files.

1.5.1 Installation Package for Microsoft Windows PC

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

To install and run COSY INFINITY in Windows, an x86 or x64 based PC running Microsoft Windows is required. COSY INFINITY has been verified on most flavors of Windows 7, in particular in 64 and 32 bit modes, as well as the earlier Vista and XP Professional and also later Windows.

To install COSY INFINITY for Windows, download and run the COSY INFINITY Installer Package for Microsoft Windows, `wincosy10.0.exe` (or a later version with the similar name). COSY INFINITY will be installed to the directory/folder that you specify, by default called "COSY 10.0" (or a later version with the similar name). The folder contains the executable program and the execution support files, and also a readme file with a quick start guide. The installer will also associate all COSYScript files with the "`.fox`" extension on your system to allow running COSY INFINITY with the COSY GUI interface as described below, which requires Java to be installed ahead of time. In addition, the environment variable COSY10.0PATH (or a later version with the similar name) is set appropriately in case you need to find COSY INFINITY 10.0 from the command line.

Running COSY INFINITY

There are several ways to run the COSY INFINITY system program. There are a few convenient ways set up by the installer, which offer the COSY GUI (graphical user interface) environment. For more information, please refer to Section 1.7.

- Double click any COSYScript file with the file extension "`.fox`".
- Right click any COSYScript file with the file extension "`.fox`", and select "Run" (or "Run with COSY INFINITY").
- From the Start Menu, start the program "Run COSY 10.0" (or a later version with the similar name), which opens a window prompting you to "Select FOX file to run". Use this window to navigate to specify a COSYScript file with the file extension "`.fox`".

1.5.2 Installation Package for Mac OS X

An executable program for Mac OS X 10.5 and higher produced by the Intel Fortran Compiler linked with the AquaTerm graphics library is available, and its usage is recommended compared to user compilation.

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 INFINITY on Mac OS X, an Intel-based Apple Mac running Mac OS X 10.5 or higher is required. So far COSY INFINITY has been successfully tested on Mac OS X 10.6.5. It is not possible to run COSY INFINITY on older PowerPC based systems.

To use COSY INFINITY 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. However, 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 INFINITY 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 INFINITY for Mac comes with a little extension for Xcode that enables syntax highlighting for editing COSYScript files (with the extension "`.fox`") with Xcode.

To install COSY INFINITY for Macintosh, simply download the Installer Package, `maccosy10_0.zip` (or a later version with the similar name), and follow the instructions provided by the installation program. This will install COSY INFINITY, AquaTerm and the COSYScript language plug-in for Xcode.

A folder called COSY INFINITY 10.0 (or a later version with the similar name) 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 INFINITY. This program will automatically be associated with `.fox` files.

The 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.

Running COSY INFINITY

There are several ways to run the COSY INFINITY system program. There are a few convenient ways set up by the installer, which offer the COSY GUI (graphical user interface) environment. For more information, please refer to Section 1.7.

- Double click any COSYScript file with the file extension "`.fox`". This requires to set "Run COSY" to be the default application to open `.fox` files.
- Right click any COSYScript file with the file extension "`.fox`", and select "Run COSY" from the "Open With" pop-up menu.
- Start the program "Run COSY" from Application folder. Click on the "Run COSY" icon in the Dock, which prompts you to navigate to specify a COSYScript file with the file extension "`.fox`", or drag a COSYScript file onto the dock icon.

1.5.3 Linux/UNIX-like Systems

An optimized executable program for Linux produced by the Intel Fortran Compiler is available, packed in `lincosy10_0.tbz` (or a later version with the similar name), at the COSY INFINITY download site. This executable is built on CentOS 5.5 that is closely related to a free version of RedHat Linux. Due

to the general dependency issue to platforms in various Linux flavors, the executable is not linked to the graphics library PGPLOT.

To install the COSY INFINITY executable program, copy the executable program "`cosy`" from the package into one of your local directories for executables. Typically this would be `/usr/local/bin`. This can be achieved by executing the command

```
cp cosy /usr/local/bin
```

To **run COSY INFINITY**, just type

```
cosy
```

The COSY INFINITY executable is a 32 bit binary. If your system is purely 64bit and not equipped to run 32bit code, you may have to install the packages "`glibc.i686`" and "`libgcc.i686`" before running the executable program "`cosy`". Typically, this problem manifests itself with an error message such as

```
/lib/ld-linux.so.2:  bad ELF interpreter:  No such file or directory
```

In RedHat based Linux flavors, you can install the required packages using

```
yum install libgcc.i686 glibc.i686
```

If all you see when running the executable program "`cosy`" is the message "`killed`", this indicates that you do not have enough memory available to run this pre-compiled version of COSY INFINITY executable program. COSY INFINITY requires 2GB of system memory to be available to run successfully. Please increase the amount of memory available to COSY INFINITY to run the pre-compiled version of COSY INFINITY executable program.

Should there be still trouble installing or running COSY INFINITY on your flavor of Linux, we recommend trying the latest version of Ubuntu Linux, which is what we use to test our releases.

If this executable program works for your need on your local Linux platform, please refer to Section 1.7 for running COSY INFINITY.

Instead of using the executable of the Linux installation package, the user may need to install COSY INFINITY by compiling the COSY INFINITY Fortran source files. Besides some cases when the above executable program is not compatible with your local systems, typical situations arise when linking to PGPLOT is desired, or when it is necessary to install COSY INFINITY for parallel computations. While some Linux systems have a pre-installed PGPLOT library, it may be necessary to install a local PGPLOT library by compiling the PGPLOT Fortran source files. Section 1.5.6 provides the instructions to install COSY INFINITY with and without linking PGPLOT, and the instructions of installing PGPLOT are provided in Section 1.5.7. For installing COSY INFINITY for parallel environments, please see Section 1.5.8.

1.5.4 Source Files

If it is necessary to install a COSY INFINITY executable program by yourself without using the installation packages for Microsoft Windows, Mac OS X, and Linux, explained in the previous subsections, the Fortran source files and some installation support files such as `Makefile` are available at the COSY INFINITY download site.

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. **foxfit.f** contains the package of nonlinear optimizers. **foxgraf.f** contains the available graphics output drivers, which are listed in Section 5.2. The foxgraf.f file available at the COSY INFINITY download site is prepared without linking to PGPLOT, GrWin, AquaTerm libraries. If local PGPLOT, GrWin, AquaTerm libraries are available, the desired libraries can be linked after modifying the source file foxgraf.f. See Section 1.5.5 for modifying foxgraf.f, and see Section 5.2 regarding the graphics output drivers.

All the Fortran parts of COSY INFINITY are written in standard ANSI Fortran 77. However, certain aspects of Fortran 77 are platform dependent; in particular, this concerns file handling, command line handling, and the CPU time measurement. As of December 2010, all commonly available Fortran compilers have identical file handling. The following compilers have been verified for compatibility with the COSY INFINITY 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

  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. [12]
- Pathscale Fortran Compiler for AMD Opteron
- IBM XL Fortran for IBM RS/6000, including MPI parallel features

The distributed four Fortran source files are compatible with the above compilers. In general, the default compiler optimization is recommended. According to our experiences and studies related to speed and reliability particularly for verified computations [12], we are currently recommending to use the Intel Fortran Compiler rather than GNU Fortran compilers. To obtain ideas for suitable compiler options, please refer to the report [12] and the descriptions in the distributed Makefile "`Makefile`" (available at the COSY INFINITY download site aimed for Linux).

Should there be additional problems, a short message to us would be appreciated in order to facilitate life for future users on the same system.

1.5.5 Conversion of a Source File Using VERSION

There are some situations when some of COSY INFINITY Fortran source files have to be adjusted for specific purposes, for example linking to the PGPLOT graphics library, or installing the MPI version of COSY INFINITY for parallel computations. The necessary conversion can be accomplished using the small program VERSION.

First, install the program VERSION using the Fortran source file `version.f`, which is available at the COSY INFINITY download site. In Linux/UNIX, the following command will install VERSION using the Intel Fortran:

```
ifort version.f -o VERSION
```

Example of VERSION to Convert foxgraf.f for PGPLOT:

This example shows how to convert the standard `foxgraf.f` file downloaded from the COSY INFINITY download site to the PGPLOT linking version. In the terminal (shell, console) window, start the program VERSION by typing "`version`", and supply the following as the program prompts for your input.

- the original file name "`foxgraf.f`"
- the new file name as a result of VERSION conversion (any name is OK, below foxgrafPGP.f is given as a mere example)
- the current ID name (nothing, because the file foxgraf.f is the original standard version)
- the new/target ID name for the conversion "`*PGP`"

Below, you see what is displayed in the console screen. When the conversion is successfully completed, the program ends with the message, "`The VERSION change finished.`"

```
     ****************************************************
     *                                                  *
     *             UTILITY PROGRAM  VERSION             *
     *                                                  *
     * This program changes the type of machine/system. *
     * The current COSY INFINITY system supports        *
     * NORM MPI FACE RND and PGP GRW AQT.               *
     * See the User's Guide and Reference Manual.       *
     *                                                  *
     ****************************************************

 GIVE OLD FILENAME:
foxgraf.f

 GIVE NEW FILENAME:
foxgrafPGP.f

 SPECIFY ID OF CURRENT VERSION (MUST START WITH * OR C):
 Examples: *PGP *GRW *AQT, and *NORM *MPI *FACE *RND



 SPECIFY ID OF NEW VERSION (MUST START WITH * OR C):

*PGP
 The VERSION change finished.
```

1.5.6 Installation by Fortran Compilation

To install the C++ and the F90 Interface Packages, please refer to Section 7 and Section 8. Below, we describe the procedures how to install COSY INFINITY by compiling the Fortran source files. Typically such cases may arise for Linux systems and UNIX-like systems, so the descriptions below assume Linux/UNIX-like systems. Depending on the local platform, some details will have to be adjusted.

Should there be any difficulties, we would appreciate hearing about them for a verification of the master version. Should you plan to install COSY INFINITY system programs on yet another system which requires changes, please send us a complete description about the changes for inclusion in the master version.

Compiling COSY INFINITY without PGPLOT

The four Fortran source files

`foxy.f`, `dafox.f`, `foxfit.f`, `foxgraf.f`

mentioned in Section 1.5.4 have to compiled and linked. A makefile "`Makefile`" for the Intel Fortran compiler is available at the COSY INFINITY download site. When the executable program `cosy` is successfully produced by the `make` process, proceed to Section 1.7 for running COSY INFINITY.

Compiling COSY INFINITY with PGPLOT Linked

See the next section 1.5.7 about the graphics library PGPLOT, and have the PGPLOT library prepared. The procedures to compile and link COSY INFINITY Fortran source files with PGPLOT below assume that the X Window System (X-Windows, `X11`) is available on your local UNIX based machine, to which PGPLOT graphics is going to be output.

1. **Conversion of foxgraf.f**: The standard `foxgraf.f` file, as downloaded from the COSY INFINITY download site, is prepared without linking to PGPLOT. So, the file `foxgraf.f` has to be modified using the program VERSION; please follow the instructions in Section 1.5.5.
2. **Compiling COSY INFINITY Fortran source files with PGPLOT:** Modify the makefile "`Makefile`" available at the COSY INFINITY download site to activate the "`LIBS=`" description to use PGPLOT.

When the executable program `cosy` is successfully produced by the `make` process, proceed to Section 1.7 for running COSY INFINITY. The Beam Physics demo program `demo.fox`, available at the COSY INFINITY download site, is a good test case to check if the PGPLOT interactive graphics output works well. Just before running `demo.fox`, the COSY Beam Physics library program `cosy.fox` has to be run, and the data file `SYSCA.DAT` has to be placed in the executing directory; See Section 1.7.6 for running COSY INFINITY for `cosy.fox` and `demo.fox`.

1.5.7 Preparation of the PGPLOT Library

The PGPLOT Graphics Subroutine Library is a graphics package copyrighted by California Institute of Technology, and is written mostly in standard Fortran 77. As of July 2013, the latest release of PGPLOT

is Version 5.2.2 of February 2001. Some Linux systems have a pre-installed PGPLOT library; in such a case the installation of COSY INFINITY linking with PGPLOT is fairly easy. If it is not the case, one needs to install the PGPLOT library using a Fortran compiler ahead of compiling and linking COSY INFINITY Fortran source files. Even though this adds an extra step in the task of installing COSY INFINITY, due to the high quality interactive graphics outputs, namely the crisp appearance and quick response, we consider it worth the extra effort when no other interactive graphics packages such as GrWin and AquaTerm are readily available. When there is no need to produce interactive graphics outputs for a particular machine, the user may not want to be bothered to link PGPLOT to a COSY INFINITY executable program on the machine, as there are various other graphics output alternatives in COSY INFINITY. Please refer to Section 5.2 for other graphics output options; in particular, the PDF and the PS graphics drivers offer high quality graphics output by producing small size files, and doing it fast.

Using a Pre-Installed PGPLOT Library

Some Linux systems have a pre-installed PGPLOT library. The availability seems to differ from time to time and depends on each platform.

According to the information supplied by Ravi Jagasia and Alexander Wittig in 2009, on Ubuntu one can check if a pre-installed PGPLOT library exists in your machine as follows. Using the Synaptic Package Manager located under "System → Administration", search for PGPLOT to find the package `pgplot5` as a package either installed or to be installed. Alternatively, one can use the command:

```
sudo apt-get install pgplot5
```

In either case, you will need root access. This will provide the library file `/usr/lib/libpgplot.a`.

If a PGPLOT library is not pre-installed in your machine, you may want to search it on the web with keywords "pgplot5", "download", and a suitable name of the Linux flavor.

Please see item **9)** below in "PGPLOT Library Installation" for some necessary environment adjustments. Then, follow the instructions in Section 1.5.6, "Compiling COSY INFINITY with PGPLOT Linked".

Compiling PGPLOT Source Files for a PGPLOT Library

The PGPLOT source package (`pgplot5.2.tar.gz`) is available at

```
http://www.astro.caltech.edu/~tjp/pgplot/
```

Please follow the information and the instructions provided there, especially "installation instructions" for "UNIX (all varieties)" available currently as of 2013 at

```
http://www.astro.caltech.edu/~tjp/pgplot/install.html
```

which is the same instructions written in the file `install-unix.txt` that is included in the PGPLOT source package. Even though some topics are outdated since the instructions are as of 1997, the instructions supplied by the original PGPLOT distributor are basic.

Please refer to the short summary "PGPLOT Library Installation" below for the installation instructions and some necessary adjustments. When the PGPLOT library is successfully created, please proceed to Section 1.5.6, "Compiling COSY INFINITY with PGPLOT Linked". If it cannot be accomplished, it is still possible to link PGPLOT to COSY by compiling necessary PGPLOT source files and directly

linking together with COSY's compiled objective files. Please see "Compiling and Linking PGPLOT to COSY Without Creating a PGPLOT Library" below (page 18).

PGPLOT Library Installation

This is a short summary on how to install the PGPLOT library on Linux. For the simplicity, specific names are given below for the directories, which you may adjust depending on your local situation. Performing the operations below as root (super-user, `su`) will simplify the task.

1) Download the PGPLOT source package `pgplot5.2.tar.gz` from the web site of the PGPLOT distributor at

```
http://www.astro.caltech.edu/~tjp/pgplot/
```

If the above site is not reachable, the package may possibly be obtained from us.

2) Create a directory for the final PGPLOT library storage.

```
mkdir /usr/local/pgplot
```

Also, prepare a directory for the PGPLOT distribution source storage `/usr/local/src/pgplot/`. If the directory `/usr/local/src/` does not exist, create it.

```
mkdir /usr/local/src
```

3) Unpack the PGPLOT source package `pgplot5.2.tar.gz` in the directory `/usr/local/src/` so that the contents are stored in the directory `/usr/local/src/pgplot/`. Some modern unpacking programs may do this easily. Otherwise, type the following commands. The options "`-xvzf`" in the `tar` command may need to be given as "`xvzf`" without "`-`" depending on your local system.

```
cp pgplot5.2.tar.gz /usr/local/src
cd /usr/local/src
tar -xvzf pgplot5.2.tar.gz
```

4) Copy the file `drivers.list` from `/usr/local/src/pgplot/` to `/usr/local/pgplot/`, and edit the file in `/usr/local/pgplot/`. Remove comments for all the necessary devices; choose Color PS (four of `PSDRIV`), and X Windows (two of `XWDRIV`) in addition to `NULL` (`NUDRIV`) of the default.

5) In `/usr/local/pgplot/`, create a makefile by typing the command for the "`makemake`" program as follows.

```
../src/pgplot/makemake  /usr/local/src/pgplot  linux g77_gcc
```

This creates the file `makefile` for `g77` and `gcc` in the directory `/usr/local/pgplot/`. The file `makefile` has to be modified to be used for the Intel Fortran `ifort` or newer GNU Fortran like `gfortran` rather than `g77`. As a general rule, the Fortran compiler to be used for the process described in Section 1.5.6 should be used here.

6) Edit the file `makefile` of the previous step **5)**. The following instructions are based on the information supplied by Markus Neher in 2009 and the PGPLOT installation guide file `pgplot_quick.txt` written by the team developing LORENE and available at

`http://www.lorene.obspm.fr/prerequisites.html`

under the topic regarding PGPLOT. Markus Neher had installed PGPLOT for COSY INFINITY on a 64bit SuSE 11.0 platform, testing to use both `ifort` and `gfortran` (gcc 4.3).

6.1) When using `ifort`, replace "`FCOMPL=g77`" in line 25 by "`FCOMPL=ifort`". Go to **6.3)**.

6.2) When using `gfortran`, replace "`FCOMPL=g77`" in line 25 by "`FCOMPL=gfortran`", and also replace "`FFLAGC=-u -Wall -fPIC -O`" in line 26 by

"`FFLAGC=-ffixed-form -ffixed-line-length-none -u -Wall -fPIC -O`".

6.3) The information here is supplied by M. Neher, and further supplemented by Ravi Jagasia and Alexander Wittig in 2009; some of the details may need to be adjusted to the specifics of your system.

Instead of linking PGPLOT to the shared `f2c` library, PGPLOT must be linked to the static library. Assuming that the static library `libf2c.a` is located in the directory `/usr/lib64/`, replace "`-lf2c`" in lines 48-51 by "`/usr/lib64/libf2c.a`". If you are using a 32 bit system, you should locate the file in `/usr/lib32/` or `/usr/lib/`. In some cases, you can opt to not change this line and instead install the package `libf2c2-dev` with the package manager.

7) Type "`make`" in the directory `/usr/local/pgplot/`.

In the end, only the next four files have to be in the directory `/usr/local/pgplot/`. Even if the `make` process does not complete according to the descriptions in the file `makefile`, as far as these four files are created, they are suffice for PGPLOT to be linked to COSY INFINITY.

`libpgplot.a`

`grfont.dat`

`rgb.txt`

`pgxwin_server` (or `pgxwin_server.exe`)

If "`make`" doesn't work to create some of the files above, try "`make libpgplot.a`" etc. individually. `rgb.txt` exists in the directory `/usr/local/pgplot/` before executing "`make`".

R. Jagasia and A. Wittig commented for some cases such as compiling PGPLOT in Ubuntu in 2009: Some additional packages may be needed, which are not installed by default, for example the package `libx11-dev`; these can be installed via the package manager.

8) Clean up the directories by typing "`make clean`" in the directory `/usr/local/pgplot/`, and further delete all unnecessary files.

9) Set the environment parameters. This differs a lot depending on the system. In general, each end user has to make the necessary adjustments.

a) `bash` shell – this is to be added in `~/.bashrc`

```
export PGPLOT_DIR="/usr/local/pgplot"
export LD_LIBRARY_PATH="/usr/local/pgplot":$LD_LIBRARY_PATH
```

b) CygWin

Add `/usr/local/pgplot` to the `PATH` list, for example in the file `/etc/profile` .

Compiling and Linking PGPLOT to COSY Without Creating a PGPLOT Library

When a PGPLOT library cannot be created by compiling the PGPLOT source files, it is still possible to compile PGPLOT source files to be linked directly together with objective files of COSY's Fortran source files. This approach is based on a suggestion made by Shashikant Manikonda in 2006. Most of the steps described above, "PGPLOT Library Installation", apply here, though there is no need to operate as root (super-user, `su`) and specific directory names are not necessarily to be used.

Follow the above steps **1)** through **5)**, though you don't have to use the same directory names. Compile the following PGPLOT source files, then link the resulting objective files together with the objective files of Fortran source files of COSY INFINITY.

- **PGPLOT source files to be compiled and linked**
  - All the Fortran source files in the directory `pgplot/src/`. Note that there are two include files in the directory.
  - All the Fortran source files in the directory `pgplot/sys/`
  - Two C source files `grdate.c` and `gruser.c` in the directory `pgplot/sys/`
  - Two Fortran source files `nudriv.f` and `psdriv.f` in the directory `pgplot/drivers/`
  - One C source file `xwdriv.c` in the directory `pgplot/drivers/`
  - A Fortran source file `grexec.f` in the directory that executes "`makemake`" in the step **5)**, and this file is a result of "`makemake`"
- **COSY INFINITY Fortran source files to be compiled and linked**
  `foxy.f`, `dafox.f`, `foxfit.f`, `foxgraf.f`
  `foxgraf.f` has to be converted for PGPLOT using the program VERSION following the instructions in Section 1.5.5.
- **Other PGPLOT files necessary to have in the resulting `cosy` executing directory**
  - The ASCII database file `rgb.txt` in the directory `pgplot`
  - The binary PGPLOT font file `grfont.dat` as a result of "`make`" in the step **6)**. See "`makefile`" for PGPLOT. It may be possible to obtain the file from some other machines.
  - `pgxwin_server` in the step **7)** may be needed. This can be created as a result of "`make`" in the step **6)**.
- **Compiler options and linking**
  Please see step **6)** and the descriptions in "`makefile`" for PGPLOT and in "`Makefile`" for COSY INFINITY. In the "`LIBS=`" description in the COSY makefile "`Makefile`", the "`pgplot`" items are not needed, but the "`X11`" items have to be kept.

1.5.8 Installation for Parallel Environments

COSY INFINITY provides native routines that interface with MPI for parallel processing. This is useful for machines with multiple cores, or for computation on clusters. At this point, COSY INFINITY 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.

For the MPI version of COSY INFINITY, prepare the four Fortran source files

foxy.f, dafox.f, foxfit.f, foxgraf.f

(see Section 1.5.4) as follows. Download the standard COSY INFINITY Fortran source files from the COSY INFINITY download site. The MPI supports have to be activated by converting these files to the MPI version.

The files foxy.f and dafox.f must be converted from ***NORM** to ***MPI** using VERSION, while foxgraf.f and foxfit.f can remain the same. See Section 1.5.5 on how to use VERSION. Specify ***NORM** and ***MPI** as the current ID and the new ID, then VERSION un-comments all the lines that contain the string ***MPI** in columns 1 to 4, and comments all the lines containing the string ***NORM** in columns 73 to 80. The conversion of the files can be done on any machine. If done on a local machine, transfer the converted files to the cluster machine.

On the cluster machine, compile the four Fortran source files with the appropriate compiler options. This should be done with the compiler wrapper function "`mpif77`" which we recommend having made with the Intel Fortran Compiler. If you plan to perform verified computations, we recommend you to contact us first before proceeding. To compile to obtain an MPI version of COSY INFINITY executable program, `mpif77` can be used in the Makefile as the Fortran compiler instead of the usual Fortran compiler.

When the executable program `cosy` is successfully produced by the `make` process, proceed to Section 1.7 and Section 1.7.4 for running COSY INFINITY.

1.6 Memory Usage and Limitations

COSY INFINITY 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 INFINITY requires approximately 200MB of physical memory, and the ultimate size of the executable process depends on the amount of memory being allocated within COSY. The executables come pre-configured for a maximum size of a little under 2GB so as to be compatible with the limitations of the address space of 32 bit environments. As of 2013, this limitation exists in all flavors of Windows, even the 64bit versions, and Mac OS X. Should this be not enough for certain large applications, the maximal memory available for allocation can be increased by changing the parameter LMEM in all occurrences in foxy.f, dafox.f and foxgraf.f to a higher value, limited only by the underlying computational environment. For purpose of estimating the final size, increasing LMEM by 1 increases the maximally required memory by 12 bytes.

1.7 How to Run COSY INFINITY

Programs written in COSYScript with the file extension "`.fox`" can be compiled and executed by the COSY INFINITY system executable program obtained above. First, we use a brief demo program `briefdemo.fox` as an example case, which shows various COSY data types. The program file `briefdemo.fox` is available at the COSY INFINITY download site.

There are several ways to execute the COSY INFINITY system program, also depending on the platform.

1.7.1 Windows or Mac Installer Users

When using the installation packages for Microsoft Windows and Mac OS X to install the COSY INFINITY system executable program, the installers set convenient ways to run COSY INFINITY. They are prepared to be able to use the COSY GUI (graphical user interface) environment. The instructions are provided in the associated readme files in the installer packages. Please also refer to the respective sections; Section 1.5.1 on page 9 for Microsoft Windows, Section 1.5.2 on page 9 for Mac OS X. For the details on COSY GUI executions, please refer to Section 1.7.5 on page 21.

1.7.2 Execution with Input Query

This execution method applies to Linux/UNIX-like systems, including Mac OS X.

In the terminal (shell, console) window, just type "`cosy`" to execute the COSY INFINITY system program. Depending on how your program execution environment is set, you may need to type in a different way such as "`./cosy`", "`cosy.exe`", or "`a.out`". When the COSY INFINITY system program properly starts, the console screen displays the title of the COSY INFINITY system, and it asks you for a file name with extension `.fox`:

```
GIVE SOURCE FILE NAME WITHOUT EXTENSION .FOX
```

At this point you type "`briefdemo`" (without the quotation marks, just nine characters). If you make a mistake, it will prompt you again for a file name, and suggests the previous one. From now on the input works like a line editor: You can replace any erroneous characters by typing the proper ones underneath. After having entered the name successfully, you will see the following message.

```
--- BEGINNING COMPILATION
--- BEGINNING EXECUTION
```

After this, the program executes COSYScript inputs written in `briefdemo.fox`.

Upon this execution, the COSYScript file name "`briefdemo`" (without the quotation marks, just nine characters) is recorded in the file `foxyinp.dat`. At the next execution of COSY INFINITY, the file name "`briefdemo`" is suggested for the input source file. If you intend to run the same COSYScript file, in this case `briefdemo.fox`, just hit the return key to confirm the file name instead of typing the name again.

1.7.3 Single Line Execution

This execution method applies to Linux/UNIX-like systems, including Mac OS X.

In the terminal (shell, console) window, the COSY INFINITY system program can be executed by one command line mode by giving the COSYScript file name:

```
cosy briefdemo.fox
```

The file extension "`.fox`" can be omitted in this mode, thus the following works as well:

```
cosy briefdemo
```

Regarding the program execution environment, the same caution applies as described in the beginning of the previous section 1.7.2.

When the COSY INFINITY system program properly starts, the console screen displays the title of the COSY INFINITY system, and you will see the following message.

```
--- BEGINNING COMPILATION
--- BEGINNING EXECUTION
```

After this, the program executes COSYScript inputs written in `briefdemo.fox`.

1.7.4 Running COSY INFINITY for Parallel Computations

Normally Linux systems are employed to operate parallel computation environments (high performance computation systems), so the explanations below assume Linux systems and other conventional properties of the system. Since a high performance system often has its specific ways to operate parallel computations, please consult the system administrators for specific instructions.

Through the "`mpirun`" command, specify the MPI version of COSY INFINITY executable program (see Section 1.5.8 for preparing such a COSY INFINITY executable program) to be run. Using the single line execution mode described in the previous section 1.7.3, the typical `mpirun` command to be typed in the terminal window would be

```
mpirun -n <NP> ./cosy <filename>
```

assuming the MPI version of COSY INFINITY executable program `cosy` is located in the current command operating directory. <`NP`> is the number of requested processes, and <`filename`> specifies the COSYScript file (with the file extension "`.fox`").

On high performance systems with strict computation time management, the **PWTIME** command is useful to monitor CPU time being consumed.

When performing Beam Physics computations in parallel environments, an execution for `cosy.fox` to produce the binary file `COSY.bin` should be operated using only one process. Please see Section 1.7.6 for running COSY INFINITY for Beam Physics computations.

1.7.5 COSY GUI Execution

To utilize the COSY GUI (graphical user interface) functionality, explained in Section 6, the platform independent COSY GUI Java program file `COSYGUI.jar` is necessary, which is available at the COSY INFINITY download site or included in COSY INFINITY installation packages. In order to run the Java GUI program, you must have Java 8 or higher installed. If you do not have Java installed already, you can get Java for free at

```
http://www.java.com
```

There are several COSY GUI example files available at the COSY INFINITY download site:

- **guidemo.fox**: An example of how to use all COSY GUI facilities in a simple program. This program uses the picture file `coffee.png`, also available at the COSY INFINITY download site.
- **guielements.fox**: An overview over all COSY GUI elements and what they look like
- **briefdemo_basicgui.fox**: A variation of briefdemo.fox, using basic COSY GUI facilities
- **briefdemo_fullgui.fox**: A variation of briefdemo.fox, using advanced COSY GUI facilities, with full, manual adjustments

The COSY INFINITY installers for Microsoft Windows and Mac OS X set the COSY GUI execution environment. The user using those installers, please refer to the instructions in the associated readme files in the installer packages and the respective sections; Section 1.5.1 on page 9 for Microsoft Windows, Section 1.5.2 on page 9 for Mac OS X.

For Linux systems, you may install Java using the Linux distribution's package manager. Please refer to your Linux documentation for further instructions on installing Java on your system. Once Java is properly installed, run the COSY GUI Java program to execute a COSYScript file, for example `guidemo.fox`, by typing as follows.

```
java -jar COSYGUI.jar guidemo.fox
```

Depending on your Linux desktop environment, you can either start the GUI by double clicking the `COSYGUI.jar` file, or using the command line.

The Java GUI tries to find the COSY INFINITY executable program "`cosy.exe`" (Windows) or "`cosy`" (Mac OS X, Unix) to use by searching the following locations in the following ordering.

1. Location of the COSYScript file (with the file extension "`.fox`") to be executed
2. Location of the `COSYGUI.jar` file

In order to use a user self built COSY INFINITY executable program generated by Fortran compilation, one can simply copy the executable program into the same directory as the COSYScript `.fox` file to be executed. Then COSY INFINITY can be executed by the methods provided by the installer, automatically using the intended COSY INFINITY executable program.

1.7.6 Running COSY INFINITY for Beam Physics Computations

There are the Beam Physics programs written in COSYScript called `cosy.fox` and `demo.fox`, available at the COSY INFINITY download site. `SYSCA.DAT`, also available at the COSY INFINITY download site, is a data file for the computation of fast fringe field approximations (fringe field mode 2). Some of example programs in `demo.fox` use this mode.

For the Beam Physics computations, you first have to run the COSY INFINITY system executable program for the COSYScript file `cosy.fox`. When the program starts properly for `cosy.fox`, following the console screen displaying the title of the COSY INFINITY system, you will see the next message.

```
--- BEGINNING COMPILATION
--- BIN FILE WRITTEN: COSY
```

After this, the program terminates. There is now a binary file `COSY.bin`, which contains a compiled code of `cosy.fox`, and this is used via the `INCLUDE` command in all Beam Physics user input.

Whenever you start using a new COSY INFINITY executable program (due to a newer version of COSY INFINITY or using a new computer or whatever the reason is!!), **you have to run the file `cosy.fox` for the purpose of updating the binary file `COSY.bin`**. Only then it will be compatible with the new COSY INFINITY executable program.

The file `demo.fox` contains a set of user inputs written in COSYScript and also demonstrates most of COSY INFINITY's Beam Physics features. As an example, let us execute `demo.fox`. The COSYScript description of the file starts with the `INCLUDE` command:

```
INCLUDE 'COSY' ;
```

This reads the contents of the binary file `COSY.bin` in. When the program starts properly for `demo.fox`, following the console screen displaying the title of the COSY INFINITY system, you will see the next message.

```
--- BEGINNING COMPILATION
--- BIN FILE READ: COSY
--- BEGINNING EXECUTION
```

The display of the demo title "`COSY INFINITY Beam Physics Demos`" and the demo menu will follow, which is the starting performance of the COSYScript inputs written in `demo.fox`.

For Beam Physics computations, beyond the description in this section, please refer to the Beam Physics Manual of COSY INFINITY [7].

1.8 Syntax Changes

With very minor exceptions, version 10.0 is downward compatible to the previously released versions of COSY INFINITY, and any user deck for version 6 and higher should run under versions 10.0 and higher. However, Taylor models are not supported in version 10.

1.9 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
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 **RE** and are initialized to 0. Thus, the following fragment declares two variables `X` and `Y` with enough space for a single double precision number and initializes them to 1 and $1/e^3$, respectively.

```
VARIABLE X 1 ; VARIABLE Y 1 ;
X := 1 ;            {Assigns value 1 to variable X}
Y := EXP(-3) ;
```

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 **CM**. The following two fragments each create a variable `Z` and initialize it to $z = 2 - 3i$. Note that the variables `Z` and `I` have to be declared with enough space to hold two double precision numbers.

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

or

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

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:

```
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 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.

```
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)

X := RE(V) ;          {Computes the arithmetic mean of the entries of V}
```

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

2.3 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 procedure **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`.

```
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.

```
VARIABLE D2 100 ; VARIABLE DI 100 ;
D2 := D%2 ;                     {Assigns D2 to be the DA vector of the partial
                                 derivate 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.

2.4 Taylor Models (RDA Objects)

Taylor model variables [13] [15] [14] should be created evaluating expressions with elementary Taylor models. The latter can be created with the intrinsic procedure **TMVAR** 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 $(2, 5)$ to the variable D.

```
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.

Note that Taylor models are not supported in the current version of COSY INFINITY.

2.5 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> ;

where <P>, <A>, and <R> are arrays, and **POLVAL** 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 or complex, 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)`.

2.6 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 [9] 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 [19] [20]. Following these prescriptions, such systems have been designed with COSY, and as predicted in the theory, the advertised nonlinearities do indeed vanish [21]. This provides confidence in the ability to compute the underlying flows properly, and again provide confidence in their correctness.

- The Taylor model data types have been verified via rather extensive tests against high-precision arithmetic packages by Corliss and Yun [10]. 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 [18]. 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 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.

3.3 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 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

```
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

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

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

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

<**procedure name**> { <expression> } ;

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

Finally, function calls have the form

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

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

3.4 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 $< \epsilon > < N_{max} > < N_{algorithm} >$ < Objective(s) > ;

Here the names denote the visible variables that are being adjusted. ϵ is the tolerance to which the minimum is requested. N_{max} 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. $N_{algorithm}$ gives the number of the optimizing algorithm that is being used. For the various optimizing algorithms, see Section 4 (page 37). < Objective(s) > are of real or integer type and denote the objective quantities, the quantities that have to be minimized. Currently only the LMDIF optimizer ($N_{algorithm} = 4$) accepts multiple objectives.

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.

3.5 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

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). Unit numbers can be associated with particular file names by using the **OPENF** and **CLOSEF** procedures, which can be found in the index.

A user contacted us in 2017 to report an incidence of a system issued error “severe: write to READ-ONLY file,...” regarding a log output file. This turned out to be caused falsely by an antivirus program. Please refer to the description on the page found in the index under “RKLOG.DAT” in the Beam Physics Manual of COSY INFINITY.

It is also possible to have binary input and output. The syntax of real number binary input and output is similar to the syntax of **READ** and **WRITE**. Use **READB** and **WRITEB** instead.

READB <expression> <name> ;

WRITEB <expression> {<expression>} ;

Files for binary input and output have to be opened and closed by using the **OPENFB** and **CLOSEF** procedures. The syntax of **OPENFB** is the same as **OPENF**.

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

For the case of formatted input of multiple numbers, this resulting string can be broken into sub strings with the operator "|" via

<string variable> | (<I1>&<I2>)

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

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

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

There are also dedicated read commands for other data types. For example, DA vectors can be read with the procedure **DAREA** (see index).

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

Depending on the momentary type of the expression, the form of the output will be as follows. Strings are printed character by character, if necessary over several lines with 132 characters per line, followed by a line feed.

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

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

Taylor models will be output in several lines, too. In addition to the first part, which has the same format as Differential Algebraic numbers, the information about the reference point and the domain, and the remainder bound 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. Graphics objects are output in the way described in Section 5.2.

As described above, each quantity in the **WRITE** command is output in a new line. To obtain formatted output, there are utilities to convert real numbers to strings, several of which can be concatenated into one string and hence output in one line. The concatenation is performed with the string operator

"&" described in Appendix A. The conversion of a real number or a complex number pair to a string can be performed with the procedure **RECST** described in Appendix A, as well as with the more convenient COSY function

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

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

ST (<real variable>)

which uses the Fortran format G23.16.

Both **SF** and **S** can be used for a complex number pair, too. In this case, the format string should specify only one Fortran number output format, which is applied to both numbers in the pair.

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

SAVE <name> ;

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

INCLUDE <name> ;

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

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

3.6 Parallel Computations

To utilize parallel computation environments, the tasks can be distributed to parallel processes using the **PLOOP** – **ENDPROOP** control structure.

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_p processes for the parallel computations, the loop must traverse each of those N_p processes. In almost every case the first expression will be 1 and the second expression will be N_p. Note that it is recommended to avoid nesting this construct. In the situation to run only a single process, **PLOOP** behaves like the **LOOP** construct.

The **ENDPLOOP** construct takes the name associated with an array variable which can be used to share information between processes. In the next example code, the processes share the information via the array X.

```
VARIABLE X 1 NP ;
PLOOP I 1 NP ;
   {user code ;}
   ENDPLOOP X ;
```

There are several utility procedures for parallel computations. **PNPRO** returns the total number of concurrent processes N_p in parallel execution, which enables to write a general purpose code instead of hard-coding any specific number of processes. **PROOT** identifies the root process. To monitor the CPU time, **PWTIME** can be called to obtain the elapsed wall-clock time, and it is useful to keep track of the execution time on machines and clusters with time allocations. See Appendix A for more explanations.

3.7 Error Messages

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

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

`$$$`: This error message denotes runtime errors in a syntactically correct user input. Circumstances under which it is issued include array bound violations, type violations, missing initialization of variables, exhaustion of the memory of a variable, and illegal operations such as division by zero.

`!!!`: This error message denotes exhaustion of certain internal arrays in the compiler. Since the basis of COSY is Fortran which is not recursive and requires a fixed memory allocation, all arrays used in the compiler have to be previously declared. This entails that in certain cases of big programs etc., the upper limits of the arrays can be reached. In such a case the user is told which parameter has to be increased. The problem can be remedied by replacing the value of the parameter by a larger value and re-compiling. Note that all occurrences of the parameter in question have to be changed globally in *all* Fortran files.

`@@@`: This message describes a catastrophic error, and should never occur with any kind of user input, erroneous or not. It means that COSY has found an internal error in its code by using certain self checks. In the 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.D0. Depending on the system COSY is run on, this will produce information about the status at the time of error. In order to be system independent, this is done by attempting to execute the computation of the root of a negative number.

4 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 33). The optimizers for this purpose are given as Fortran subroutines. For a list of currently available optimizers, see Section 4.1. Because of a relatively simple interface, it is also possible to include new optimizers relatively easily. Details can be found in Section 4.2.

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.

4.1 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 33.

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.

4.2 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:

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

The last argument, the status identifier, communicates the status of the optimization process to the 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.

5 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 operator "&" 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 Appendix A.

5.1 Simple Pictures

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

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

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

There is also a procedure that allows drawing simple curves by line segments, also supplied in cosy.fox:

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

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

5.2 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 **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,... Information on the graphics object is included as comments at the end of the file. Each polynomial graphics object has the property noted as comments.
- -11: Direct output to the low level graphics meta files pic001.dat,...
- -12: Direct PDF (Portable Document Format) output to files pic001.pdf,... Information on the graphics object is included as comments at the end of the file. Each polynomial graphics object has the property noted as comments.

- -13: Direct SVG (Scalable Vector Graphics) output to files pic001.svg,... Information on the graphics object is included as comments at the end of the file. Each polynomial graphics object has the property noted as comments.
- -14, -15: Direct STL (STereoLithography, Standard Tessellation Language) triangle output (-14: ASCII, -15: binary) to files pic001.stl,... STL describes the surface geometry of a three dimensional object by triangles without any other common graphics representation such as color.
- -20: PGPLOT output to PostScript files pic001.ps,...
- -22: PGPLOT output to LaTeX files pic001.tex,...
- -101 ... -110: PGPLOT X-Windows workstation or Windows PC windows output
- -111 ... -120: GrWin Microsoft Windows PC windows output
- -121 ... -130: AquaTerm Mac PC windows output
- -201 ... -210: GUI output to the corresponding COSY GUI window
- -1000: Dummy and there is no output

Positive unit numbers produce a low resolution (80 columns by 24 lines) ASCII output of the picture written to the respective unit, where unit 6 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.

When a graphics object is output to a file, the file name has the prefix "pic" with the number in 3 digits, followed by the corresponding file extension name. The prefix and the starting number can be changed using **GROUTF** (see index).

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

- -101 ... -110, -20, -22: PGPLOT package
  While some Linux systems have a pre-installed PGPLOT library, it may be necessary to install a local PGPLOT library. The PGPLOT Graphics Subroutine Library is freely available for download from the PGPLOT web page, which is located at http://www.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 makefile available at the COSY INFINITY download site (see Sections 1.5, 1.5.4) shows how to link to the PGPLOT library.
  The PGPLOT driver routines in foxgraf.f have to be modified. The file foxgraf.f must be converted from the standard version to the ***PGP** version using VERSION. See Section 1.5.5 on how to use VERSION, and there is an example of converting foxgraf.f for PGPLOT linking. Specify ***PGP** as the new ID, then VERSION un-comments all the lines that contain the string ***PGP** in columns 1 to 4.
- -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 to be modified. The file foxgraf.f must be converted from the standard version to the ***GRW** version using VERSION. See Section 1.5.5 on how to use VERSION. Specify ***GRW** as the new ID, then VERSION un-comments all the lines that contain the string ***GRW** in columns 1 to 4.

- -121 ... -130: AquaTerm package
  The AquaTerm Graphics Library is freely available from the AquaTerm web page, http://aquaterm.sf.net/. If linking to AquaTerm package is desired, the AquaTerm driver routines in foxgraf.f have to be modified. The file foxgraf.f must be converted from the standard version to the ***AQT** version using VERSION. See Section 1.5.5 on how to use VERSION. Specify ***AQT** as the new ID, then VERSION un-comments all the lines that contain the string ***AQT** in columns 1 to 4.

The other graphics drivers are self-contained within COSY.

5.3 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 ten 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 end with three letters identifying the task. The required routines are

1. ...BEG : Begins the picture. Allows calling all routines necessary to initiate a picture.
2. ...MOV(X,Y,Z) : Performs a move of the pen to coordinates (X,Y,Z).
3. ...DRA(X,Y,Z) : Performs a draw from the current position to coordinates (X,Y,Z).
4. ...DOT(X,Y,Z) : Performs a move of the pen to coordinates (X,Y,Z) then prints a dot at the position.
5. ...TRI(X,Y,Z) : Performs a move of the pen to coordinates (X,Y,Z) and draws a triangle with the two previous pen positions.
6. ...PLY(IA,IPST) : Draws a polynomial curve or surface of a polynomial graphics object starting at the COSY internal memory address IA. IPST identifies the status of the x, y, z position polynomials, and is -1 if any one of them is an arithmetic failure. See Subroutine TTYPLY as a simple implementation example. See Subroutines POSPLY and POSPLY0 as examples for more dedicated implementations.
7. ...CHA(STR,L) : Prints ASCII string STR with length L at the momentary position.
8. ...COL(CLR) : Sets a color specified by RGBA values if supported by the system. CLR is an array, and CLR(1), CLR(2), CLR(3), CLR(4) are for R (red), G (green), B (blue), and A (alpha for opacity) values ranging from 0 to 1. The default values are 0, 0, 0, 1, corresponding to the fully opaque black color. When A is supported by the system, A=0 means transparent thus invisible.
9. ...WID(W) : Sets the width W of the pen. The default thickness is 1.
10. ...END : Concludes the picture, by closing the picture and printing it.

The arguments X, Y, Z, CLR(4), W are DOUBLE PRECISION, and IA, IPSA, L are INTEGER, and STR is CHARACTER STR*1024. 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.

5.4 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 files pic001.dat, ... via unit -11. This meta file can be easily read by programs written by the user.

The meta file consists of a list of elementary graphics operations. Each occurrence of these ten elementary operations discussed in the previous section 5.3 and some graphics commands is output in a separate line, where the first three characters identify the command, then follows a blank, and then the parameters. The next table shows the style of lines, followed by a detailed explanation to each line.

Lines in COSY Graphics Meta Files:

```
BEG   GRAPHICS METAFILE CREATED BY COSY INFINITY
PRJ PHI THETA
ZOO X1 X2 Y1 Y2 Z1 Z2
MOV X Y Z
DRA X Y Z
DOT X Y Z
TRI X Y Z
PLY
PL1 I1 I2 C
PL1     R C
PLE
PLY   -ARITHMETIC FAILURE-
CHA   STRING
RGB R G B A
WID W
LWR IWR
EPS EXYZ ECOL
END   COSY PICTURE
```

The file starts with the line with the command "BEG", and ends with the line with the command "END".

The values X, Y, Z for the commands "MOV", "DRA", "DOT", "TRI" for **GRMOVE**, **GRDRAW**, **GRDOT**, **GRTRI** are output in the Fortran format 3E24.16. "PRJ" lists the projection angles (radian) of the last **GRPROJ** call or the default values in the Fortran format 2E24.16. "ZOO" lists the zooming box of the last **GRZOOM** call in the Fortran format 6E24.16.

For **GRPOLY**, a polynomial graphics object's output starts with "PLY", followed by "PL1", "PL2", ..., "PL7" commands each representing one monomial in the x, y, z position polynomials ("PL1", "PL2", "PL3") and the color polynomials for R, G, B, A ("PL4", "PL5", "PL6", "PL7"). Polynomial output is finished by a "PLE" command. If any one of the x, y, z position polynomials is an arithmetic failure, the "PLY" line is marked so, concluding the polynomial graphics object's output. All the non-zero coefficients of a polynomial are listed in "PL1" through "PL7" lines, each line listing one coefficient C with the exponents of the independent variables I1 and I2 in the Fortran format I2,X,I2,X,E24.16. If the polynomial is a Taylor model, the remainder bound C is listed with a mark "R" in the Fortran format 4X,'R',X,E24.16. A cubic spline curve by **GRCURV** is converted to a polynomial graphics object or a line segment.

A string by **GRCHAR** is shown as a string of characters in its actual length following "CHA" and 3 blank characters. The values R, G, B, A for the command "RGB" due to **GRCOLR** are output in

the Fortran format `4E12.8`, and the values range from 0 to 1. The value W for the command "`WID`" by **GRWDTH** is output in the Fortran format `F8.4`. IWR by **GRSTYL** is output in the line starting with "`LWR`" in the Fortran format `I2`. EXYZ and ECOL by **GREPS** are output in the line starting with "`EPS`" in the Fortran format `E24.16`.

6 Graphical User Interface

Starting with version 9.1, COSY INFINITY has built in support for building graphical user interfaces (GUIs) from within COSYScript [22]. The GUI feature requires COSY INFINITY to be run with a GUI program, such as the platform independent Java GUI program "COSYGUI.jar". Please refer to Section 1.7.5 for running the COSY GUI Java program for COSYScript files.

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 6.1) 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 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).

6.1 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.

6.2 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 **READ**s 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.

COSY graphics objects can be written to a GUI window, they will be rendered in an area within the GUI window. Graphics objects do return a value, so any write of a graphics object can be followed by a read from the same GUI window number. In a future version of COSY, this mechanism will allow reading of the coordinates of a mouse click within a graphics picture.

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>.

GUISET <unit> <nr> <value> ;

Use this command to update the value of a component in the currently displayed window without closing and reopening the window. 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 (whose names start with "Read" and graphics objects) starting with 1. You can update graphics the same way, specifying a previously drawn graphics object as nr and a new COSY graphics object for value.

6.3 GUI Command Reference

Tables 1, 2 list 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 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. Following Tables 1 and 2, we give some further remarks on specific GUI commands.

Command	**Window**	**Value**	**Arguments**
\Console Write to embedded console	hidden	No	Any number of arguments of any type
\Text Static text	hidden	No	String to be inserted
\Link Clickable URL	hidden	No	Link text URL (link text) Tooltip (URL)
\Image Static image	hidden	No	Image filename
\Line Vertical line	hidden	No	None
\Spacer Transparent element	hidden	No	Width in pixels Height in pixels (0)
\Button Push button	hidden	No	Text on button 1 - default button, 0 - otherwise (0) Tooltip (none)
\ReadCheckbox Checkbox	hidden	Yes	Text next to checkbox (none) 1 - selected, 0 - not selected (0) Tooltip (none)
\ReadOption Radio button	hidden	Yes	Text next to radio button (none) 1 - selected, 0 - not selected (0) name of button group (none) Tooltip (none)
\ReadField Unformatted input field	hidden	Yes	Initial value (none) Tooltip (none)
\ReadNumber Numerical input	hidden	Yes	Current value Minimum value Maximum value Increment ((Max-Min)/100) 1 - editable, 0 - not editable (1) Tooltip (none)
\ReadList Selection from list	hidden	Yes	List of entries separated by | Initially selected value 1 - combobox, 0 - dropdown, L - list, M - multiselect (0) Tooltip (none)
\ReadFileName File selector	hidden	Yes	Initial value (none)
\ReadProgress Progress bar	hidden	Yes	Progress in % or -1 (-1)

Table 1: Available GUI commands in COSY

Command	**Window**	**Value**	**Arguments**
\NewLine Jump to next line	hidden	No	None
\NewCell Jump to next cell	hidden	No	Width of cell (1)
\Left Set current cell's alignment to left	hidden	No	None
\Center Set current cell's alignment to center	hidden	No	None
\Right Set current cell's alignment to right	hidden	No	None
\Just Set current cell's alignment to justified	hidden	No	None
\Title Set window title	hidden	No	Window title
\Deactivate Make non-interactive	displayed	N/A	None
\Activate Make interactive	displayed	N/A	None
\Show Display window	hidden	N/A	x coordinate (center) y coordinate (center)
\Close Close and destroy window	displayed	N/A	None
\Set Set value of interactive element	displayed	N/A	Number of element Value to be set
\Focus Make this the active window	displayed	N/A	None
\PutFile Put local file in execution directory	N/A	N/A	remote filename local filename (prompt user)
\GetFile Get remote file from execution directory	N/A	N/A	remote filename local filename (remote filename)
\Debug Set GUI debug level	all	N/A	Debug level between 0 – 3
\Finish Add a button if none is there yet	hidden	No	Text on the button ('OK')

Table 2: Available GUI commands in COSY (continued)

`\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 6.4 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`
: It is not recommended to use this command directly. Use GUISET procedure instead.

`\PutFile, \GetFile`
: When the GUI is executed on a remote machine, these commands can be used to transfer files between the local machine and the remote server. If the GUI is run locally, this will simply perform a local copy operation of the files.

`\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.

6.4 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`.

6.5 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.

- `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.

There are several examples of COSY GUI COSYScript program files, available at the COSY INFINITY download site. Please refer to Section 1.7.5 for the details and how to execute them.

7 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 8) 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 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 Taylor models.

7.1 Installation

The implementation of the Cosy class is based on the Fortran 77 files which make up the core source implementation of the COSY INFINITY system. Most of the actual C++ code is automatically generated from these Fortran 77 files by the **F2C converter** [11]. For purposes of guaranteed robustness of the translation, we maintain a source version of F2C that can be downloaded from the COSY download site, which itself can be compiled with any standard C compiler leads guaranteed Consequently, use of the Cosy class requires the **F2C library** to be installed on the user's system. One generic source of F2C is located at http://www.netlib.org/f2c/ , but various dedicated binary distribution of the library for a particular combination of compiler and system libraries exist and may be preferable. For the user's convenience, the source code of said library is also available from the COSY INFINITY download site. 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.

7.2 Memory Management

The Cosy class manages its own internal memory and does not use dynamic allocation of memory by either malloc or new. 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[]`, especially since high-dimensional multi-variable Cosy objects can be very large in size. Consequently, *dynamic allocation of Cosy objects should be used cautiously or avoided.* If Cosy objects have to be created dynamically, care should be taken to delete the objects as soon as possible, or the COSY system will exhaust its internal memory.

7.3 Public Interface of the Cosy Class

In this section we describe the public interface of the Cosy class. Most of the functions and operators described in this section fall in the categories of constructor, assignment, and unary operators and have no equivalent constructs in the standard COSY INFINITY language described in Section 3. Therefore, reading this section is essential for the understanding of the C++ interface to COSY INFINITY.

7.3.1 Constructors

To allow an easy conversion of existing code from the double data type to the Cosy data type, several constructors have been defined that should accommodate this through a variety of implicit constructions. Together with the built-in type conversions of C++, this mechanism should be able to handle almost any situation correctly. 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 `len` numbers or characters. The parameter `len` is optional and defaults to 1. The object's type is initialized to **RE** with value `val`.

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

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

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

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

```
Cosy( const char *str );
```

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

```
Cosy( const Cosy& src );
```

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

```
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 `dim`. And each entry of the array has internal space for `len` numbers and is initialized to zero with type **RE**. For further details on Cosy arrays, refer to Section 7.6.

7.3.2 Assignment Operators

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

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

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

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

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

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

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

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

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

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

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

```
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++.

7.3.3 Unary Mathematical Operators

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

7.3.4 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 7.6.

```
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 7.6.

7.3.5 Printing, IO, and Streams

As indicated earlier, the code for the Cosy class is automatically derived from Fortran 77 code by using the F2C converter [11]. 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

```
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.

7.3.6 Type Conversion

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

```
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.

```
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.

```
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.

7.4 Elementary Operations and Functions

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

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

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

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

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

The standard functions defined for the Cosy class are listed in 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 <NAME> is as follows:

```
Cosy <name> (const Cosy& x);
```

where <name> is the name of the function from Appendix A. Note that for C++ use, all names have to replaced by lowercase.

7.5 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 Appendix A. Note that a "`c`" parameter stands for "`const Cosy &`" arguments; a "`v`" parameter denotes "`Cosy &`" arguments. The name of the procedure has to be supplied in lowercase.

7.6 Cosy Arrays vs. Arrays of Cosy Objects

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

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

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

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

and

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

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

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

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

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

8 The Fortran 90 Interface

The Fortran 90 interface to COSY 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.

8.1 Installation

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

The distribution contains the four Fortran 77 files that make up the COSY INFINITY system (c.f. Section 1.5 for details on how to compile these files). However, some changes have been made in the file foxy.f to enable use in the Fortran 90 module. The file foxy.f must be converted from ***NORM** to the ***FACE** version using VERSION. Specify ***NORM** and ***FACE** as the current ID and the new ID, then VERSION un-comments all the lines that contain the string ***FACE** in columns 1 to 5, and comments all the lines containing the string ***NORM** in columns 73 to 80. See Section 1.5.5 on how to use VERSION.

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.

8.2 Special Utility Routines

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

`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 8.7 for more details on the pool of temporary objects.) The command

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

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 **RE**.

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

`SUBROUTINE COSY_DESTROY <SELF>`

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

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

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 8.8 for further details on COSY arrays).

`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 8.8 for further details on COSY arrays).

`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 **RE**. Refer to Section 8.7 for more details.

`SUBROUTINE COSY_DOUBLE <SELF>`

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

`SUBROUTINE COSY_LOGICAL <SELF>`

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

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

writes the COSY variable SELF to the unit IUNIT (which defaults to 6). This function uses the same algorithms employed by the COSY procedure **WRITE** (c.f. Section 3.5).

`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.

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

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

8.3 Operations

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

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

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

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

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

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

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

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

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

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

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

Extraction .EX.		
COSY	COSY	COSY
COSY	DOUBLE PRECISION	COSY
COSY	INTEGER	COSY

Derivation .DI.		
COSY	COSY	COSY
COSY	DOUBLE PRECISION	COSY
COSY	INTEGER	COSY

8.4 Assignment

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

COSY LHS = COSY RHS

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

DOUBLE PRECISION LHS = COSY RHS

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

LOGICAL LHS = COSY RHS

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

COSY LHS = DOUBLE PRECISION RHS

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

COSY LHS = LOGICAL RHS

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

COSY LHS = INTEGER RHS

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

8.5 Functions

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

8.6 Subroutines

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

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

8.7 Memory Management

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

When a COSY object is created by using the routine COSY_CREATE, memory is allocate in the internal COSY memory. This memory is not freed until the routine COSY_DESTROY is called for this object. Moreover, since COSY's internal memory is stack based 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.

8.8 COSY Arrays vs. Arrays of COSY objects

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

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

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

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

and

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

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

9 Acknowledgements

For very valuable help with an increasing number of parts of the program, we would like to thank Meng Zhao, Weishi Wan, Georg Hoffstätter, Ralf Degenhardt, Nina Golubeva, Vladimir Balandin, Jens Hoefkens, Béla Erdélyi, Laura Chapin, Shashikant Manikonda, Youn-Kyung Kim, Pavel Snopok, Alexey Poklonskiy, Johannes Grote, Alexander Wittig, He Zhang, and Ravi Jagasia 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, and our special thanks go to Markus Neher, George Corliss and Nathalie Revol. 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 Tsuguhiro Tamaribuchi for plotting on Windows, AquaTerm for plotting on Mac OS X, PGPLOT for plotting on Linux, WeFunction icon set from www.wefunction.com for GUI icons in the COSY GUI 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

[1] M. Berz. Forward algorithms for high orders and many variables. In *Automatic Differentiation of Algorithms: Theory, Implementation and Application*, pages 147–156. SIAM, Philadelphia, 1991.

[2] M. Berz. From Taylor series to Taylor models. *AIP CP*, 405:1–20, 1997.

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

[4] M. Berz, C. Bischof, A. Griewank, G. Corliss, and Eds. *Computational Differentiation: Techniques, Applications, and Tools*. SIAM, Philadelphia, 1996.

[5] M. Berz, H. C. Hofmann, and H. Wollnik. COSY 5.0, the fifth order code for corpuscular optical systems. *Nuclear Instruments and Methods*, A258:402–406, 1987.

[6] M. Berz and K. Makino. Suppression of the wrapping effect by Taylor model- based verified integrators: Long-term stabilization by shrink wrapping. *International Journal of Differential Equations and Applications*, 10,4:385–403, 2005.

[7] M. Berz and K. Makino. COSY INFINITY Version 10.0 beam physics manual. Technical Report MSUHEP-151103-rev, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2017. See also http://cosyinfinity.org.

[8] M. Berz, K. Makino, and Y.-K. Kim. Long-term stability of the Tevatron by validated global optimization. *Nuclear Instruments and Methods*, 558:1–10, 2006.

[9] M. Berz and H. Wollnik. The program HAMILTON for the analytic solution of the equations of motion in particle optical systems through fifth order. *Nuclear Instruments and Methods*, A258:364–373, 1987.

[10] G. F. Corliss and J. Yu. Interval testing strategies applied to cosy's interval and taylor model arithmetic. In R. A. et al., editor, *Numerical Software with Result Verification*, volume LNCS 2991, pages 91–106. Springer, 2004.

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

[12] R. Jagasia and A. Wittig. Survey of FORTRAN compiler options and their impact on COSY INFINITY. Technical Report MSUHEP-090422, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2009.

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

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

[15] K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 6,3:239–316, 2003.

[16] K. Makino and M. Berz. Suppression of the wrapping effect by Taylor model- based verified integrators: Long-term stabilization by preconditioning. *International Journal of Differential Equations and Applications*, 10,4:353–384, 2005.

[17] K. Makino and M. Berz. Suppression of the wrapping effect by Taylor model- based verified integrators: The single step. *International Journal of Pure and Applied Mathematics*, 36,2:175–197, 2006.

[18] N. Revol, K. Makino, and M. Berz. Taylor models and floating-point arithmetic: Proof that arithmetic operations are validated in COSY. *Journal of Logic and Algebraic Programming*, 64/1:135–154, 2004.

[19] W. Wan. *Theory and Applications of Arbitrary-Order Achromats*. PhD thesis, Michigan State University, East Lansing, Michigan, USA, 1995. also MSUCL-976.

[20] W. Wan and M. Berz. Design of a fifth order achromat. *Nuclear Instruments and Methods*, 352, 1994.

[21] W. Wan and M. Berz. Analytical theory of arbitrary-order achromats. *Physical Review E*, 54(3):2870–2883, 1996.

[22] A. Wittig, M. Berz, and K. Makino. The COSY INFINITY graphical user interface subsystem. Technical Report MSUHEP-111101, Department of Physics and Astronomy, Michigan State University, East Lansing, MI 48824, 2011.

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.

subsequent information is automatically generated by the GENFOX syntax management system, and is current as of 31-Mar-2017.

A.1 Objects

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

RE	8 Byte Real Number
ST	String
LO	Logical
CM	8 Byte Complex Number
VE	Vector of 8 Byte Real Numbers
DA	Differential Algebra Vector
CD	Complex Differential Algebra Vector
GR	Graphics

A.2 Operators

Now follows a list of all operators available for various combinations of objects. Allowed types of the left and the right operands are shown as well as the resulting types of the operation.

For each operation, a relative priority is given which determines the hierarchy of the operations in expressions if there are no parentheses. An operation with a larger priority number has higher priority.

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

Left	Right	Result	Comment
RE	RE	RE	
RE	CM	CM	
RE	VE	VE	Add Real componentwise
RE	DA	DA	
RE	CD	CD	
LO	LO	LO	Logical OR
CM	RE	CM	
CM	CM	CM	
CM	DA	CD	
CM	CD	CD	
VE	RE	VE	Add Real componentwise
VE	VE	VE	Add componentwise
DA	RE	DA	
DA	CM	CD	
DA	DA	DA	
DA	CD	CD	
CD	RE	CD	
CD	CM	CD	
CD	DA	CD	
CD	CD	CD	

- **– (Subtraction)** (Priority: 3)

Left	Right	Result	Comment
RE	RE	RE	
RE	CM	CM	
RE	VE	VE	Subtract componentwise from Real
RE	DA	DA	
RE	CD	CD	
CM	RE	CM	
CM	CM	CM	
CM	DA	CD	
CM	CD	CD	
VE	RE	VE	Subtract Real componentwise
VE	VE	VE	Subtract componentwise
DA	RE	DA	
DA	CM	CD	
DA	DA	DA	
DA	CD	CD	
CD	RE	CD	
CD	CM	CD	
CD	DA	CD	
CD	CD	CD	

- *** (Multiplication)** (Priority: 4)

Left	Right	Result	Comment
RE	RE	RE	
RE	CM	CM	
RE	VE	VE	Multiply with Real componentwise
RE	DA	DA	
RE	CD	CD	
LO	LO	LO	Logical AND
CM	RE	CM	
CM	CM	CM	
CM	DA	CD	
CM	CD	CD	
VE	RE	VE	Multiply with Real componentwise
VE	VE	VE	Multiply componentwise
DA	RE	DA	
DA	CM	CD	
DA	DA	DA	
DA	CD	CD	
CD	RE	CD	
CD	CM	CD	
CD	DA	CD	
CD	CD	CD	

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

Left	Right	Result	Comment
RE	RE	RE	
RE	CM	CM	
RE	VE	VE	Divide Real componentwise
RE	DA	DA	
RE	CD	CD	
CM	RE	CM	
CM	CM	CM	
CM	DA	CD	
CM	CD	CD	
VE	RE	VE	Divide by Real componentwise
VE	VE	VE	Divide componentwise
DA	RE	DA	
DA	CM	CD	
DA	DA	DA	
DA	CD	CD	
CD	RE	CD	
CD	CM	CD	
CD	DA	CD	
CD	CD	CD	

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

Left	Right	Result	Comment
RE	RE	RE	
VE	RE	VE	Raise to Real power componentwise

- < **(Less Than)** (Priority: 2)

Left	Right	Result	Comment
RE	RE	LO	
ST	ST	LO	Lexicographic Ordering

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

Left	Right	Result	Comment
RE	RE	LO	
ST	ST	LO	Lexicographic Ordering

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

Left	Right	Result	Comment
RE	RE	LO	
ST	ST	LO	

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

Left	Right	Result	Comment
RE	RE	LO	
ST	ST	LO	

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

Left	Right	Result	Comment
RE	RE	VE	Concatenate two Reals to a Vector
RE	VE	VE	Append a Real to the left of a Vector
ST	ST	ST	Concatenate two Strings
VE	RE	VE	Append a Real to the right of a Vector
VE	VE	VE	Concatenate two Vectors
GR	GR	GR	Concatenate two Graphics Objects

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

Left	Right	Result	Comment
RE	RE	RE	(no effect when the 1st component is requested)
RE	VE	RE	(no effect when the 1st component is requested)
ST	RE	ST	Extract the i-th component
ST	VE	ST	Extract component range in two-vector
CM	RE	RE	Input 1: real part, 2: imaginary part
VE	RE	RE	Extract the i-th component
VE	VE	VE	Extract component range in two-vector
DA	RE	RE	Extract coefficient of 1D DA for supplied exponent
DA	VE	RE	Extract coefficient for exponents in vector
CD	RE	CM	Extract coefficient of 1D CD for supplied exponent
CD	VE	CM	Extract coefficient for exponents in vector

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

Left	Right	Result	Comment
RE	RE	DA	Diff. ($i > 0$, Result=0) or Integ. ($i < 0$) w.r.t. $x_{\|i\|}$
CM	RE	CD	Diff. ($i > 0$, Result=0) or Integ. ($i < 0$) w.r.t. $x_{\|i\|}$
DA	RE	DA	Differentiate ($i > 0$) or Integrate ($i < 0$) w.r.t. $x_{\|i\|}$
CD	RE	CD	Differentiate ($i > 0$) or Integrate ($i < 0$) w.r.t. $x_{\|i\|}$

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)

Argument	Result	Comment
RE	RE	(no effect)
ST	RE	Converts a String to Real
CM	RE	Extracts the Real part
VE	RE	Determines the average
DA	RE	Extracts constant part of DA

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

Argument	Result	Comment
RE	ST	Formatted Conversion
ST	ST	(no effect)
LO	ST	Text of the logical values True or False
CM	ST	Formatted Conversion

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

Argument	Result	Comment
RE	LO	1: True, 0: False
LO	LO	(no effect)

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

Argument	Result	Comment
RE	CM	Converts real number to complex
CM	CM	(no effect)
VE	CM	Converts two-vector with real and imaginary parts
CD	CM	Extracts constant part from Complex DA Vector

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

Argument	Result	Comment
RE	RE	(no effect)
CM	VE	Extracts real and imaginary parts in two-vector
VE	VE	(no effect)

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

Argument	Result	Comment
RE	DA	Generates the i-th component of identity DA vector
DA	DA	(no effect)
CD	DA	Extracts the Real part

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

Argument	Result	Comment
RE	CD	Generates the i-th component of identity CD vector
DA	CD	Converts DA to CD
CD	CD	(no effect)

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

Argument	Result	Comment
RE	RE	

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

Argument	Result	Comment
RE	RE	Input: length of string

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

Argument	Result	Comment
RE	RE	Input: arbitrary

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

Argument	Result	Comment
RE	RE	Input: arbitrary

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

Argument	Result	Comment
RE	RE	Input: number of components

- **LDA** Determines allocation size of DA

Argument	Result	Comment
VE	RE	Input: two-vector consisting of order, variables

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

Argument	Result	Comment
VE	RE	Input: two-vector consisting of order, variables

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

Argument	Result	Comment
RE	RE	Input: number of GR elements (Output: approximate length)

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

Argument	Result	Comment
RE	RE	
ST	RE	
LO	RE	
CM	RE	
VE	RE	
DA	RE	
CD	RE	
GR	RE	

- **LENGTH** Returns the currently used memory of an object (8 byte blocks)

Argument	Result	Comment
RE	RE	
ST	RE	
LO	RE	
CM	RE	
VE	RE	
DA	RE	
CD	RE	
GR	RE	

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

Argument	Result	Comment
RE	RE	
ST	RE	
LO	RE	
CM	RE	
VE	RE	
DA	RE	
CD	RE	
GR	RE	

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

Argument	Result	Comment
RE	RE	
ST	RE	
LO	RE	
CM	RE	
VE	RE	
DA	RE	
CD	RE	
GR	RE	

- **EXP** Computes the exponential function

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	

- **LOG** Computes the natural logarithm

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	

- **SIN** Computes the sine

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	

- **COS** Computes the cosine

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	

- **TAN** Computes the tangent

Argument	Result	Comment
RE	RE	
VE	VE	
DA	DA	

- **ASIN** Computes the arc sine

Argument	Result	Comment
RE	RE	
VE	VE	
DA	DA	

- **ACOS** Computes the arc cosine

Argument	Result	Comment
RE	RE	
VE	VE	
DA	DA	

- **ATAN** Computes the arc tangent

Argument	Result	Comment
RE	RE	
VE	VE	
DA	DA	

- **SINH** Computes the hyperbolic sine

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	

- **COSH** Computes the hyperbolic cosine

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	

- **TANH** Computes the hyperbolic tangent

Argument	Result	Comment
RE	RE	
VE	VE	
DA	DA	

- **SQRT** Computes the square root

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	

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

Argument	Result	Comment
RE	RE	
VE	VE	
DA	DA	

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

Argument	Result	Comment
RE	RE	
VE	VE	
DA	DA	

- **SQR** Computes the square

Argument	Result	Comment
RE	RE	
CM	CM	
VE	VE	
DA	DA	
CD	CD	

- **ERF** Computes the real error function erf

Argument	Result	Comment
RE	RE	
DA	DA	

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

Argument	Result	Comment
CM	CM	
CD	CD	

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

Argument	Result	Comment
VE	RE	

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

Argument	Result	Comment
VE	RE	

- **ABS** Computes the absolute value

Argument	Result	Comment
RE	RE	
CM	RE	
VE	RE	Determines the sum of absolute values of components
DA	RE	Determines the max norm of coefficients
CD	RE	Determines the max of the max norms of real and imag. parts

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

Argument	Result	Comment
VE	VE	same as ABS
DA	RE	same as ABS
CD	RE	same as ABS

- **CONS** Determines the constant part of certain types

Argument	Result	Comment
RE	RE	
CM	CM	
VE	RE	Determines the largest absolute value of components
DA	RE	
CD	CM	

- **REAL** Determines the real part of certain types

Argument	Result	Comment
RE	RE	
CM	RE	
DA	DA	
CD	DA	

- **IMAG** Determines the imaginary part of certain types

Argument	Result	Comment
RE	RE	
CM	RE	
DA	DA	
CD	DA	

- **CMPLX** Converts types to complex

Argument	Result	Comment
RE	CM	
CM	CM	
DA	CD	
CD	CD	

- **CONJ** Determines the complex conjugate of certain types

Argument	Result	Comment
RE	RE	
CM	CM	
DA	DA	
CD	CD	

- **INT** Determines the integer part

Argument	Result	Comment
RE	RE	
VE	VE	

- **NINT** Determines the nearest integer

Argument	Result	Comment
RE	RE	
VE	VE	

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

Argument	Result	Comment
LO	LO	

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

Argument	Result	Comment
ST	ST	

- **LTRIM** Removes the space characters from the beginning of a string

Argument	Result	Comment
ST	ST	

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

Argument	Result	Comment
RE	RE	

A.4 Intrinsic Procedures

The following is a list of all available intrinsic 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).

- **SLEEPM** (c)
  Suspends program execution for a given duration (milli-sec).

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

- **ARGGET** (cv)
  Returns the n-th command line argument. This interfaces to the GETARG intrinsic subroutine in FORTRAN. Arguments are n and the resulting string. If the n-th command line argument does not exist, an empty string is returned.

- **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.

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

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

- **READS** (cv)
  Reads a string without attempting to convert it to RE. The arguments are the unit number and the variable name.

- **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.

- **READM** (vccccc)
  Reads arrays for a variable in the form of the COSY memory contents. The arguments are (1) the variable name (any data type), (2) the variable information (VE), (3) the length of arrays (RE), (4) the array for the COSY memory double precision part (RE array), (5) the array for the COSY memory integer part (RE array), (6) the DA parameters if DA or CD (VE); else 0 (RE). READM is meant to input the output contents by WRITEM. Refer to WRITEM. The supplied DA parameters (6) are checked for the compatibility against the current DAINI setup.

- **WRITEM** (cvcvvv)
  Writes the COSY memory contents of a variable in arrays. The arguments are (1) the variable name (any data type), (2) the variable information (VE), (3) the length of arrays (RE), (4) the array for the COSY memory double precision part (RE array), (5) the array for the COSY memory integer part (RE array), (6) the DA parameters if DA or CD (VE); else 0 (RE). The variable information (2)

consists of the data type, the length in the COSY memory, and the WRITEM version identification number. The DA parameters (6) consists of the order, the number of variables, and when weighted DA is setup, the weight factors.

- **DAINI** (cccv)
  Initializes the order and number of variables of DA or CD. 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 and CD.

- **DANOTW** (cc)
  Sets weighted order factor of each independent variable for DA and CD. 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 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 and CD vectors.

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

- **EPSMIN** (v)
  Returns the underflow threshold, the smallest positive number representable on the system. This number is determined automatically whenever DA or CD is 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 and CD. If the argument is 0 or DAINI is called, the filtering mode is turned off.

- **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** (ccccc)
  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. Arguments are the DA or CD vector, the id for the coefficient in TRANSPORT notation (for example, the id for the $x_1 x_3^2$ 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.

- **DACODE** (ccv)
  Decodes the DA internal monomial numbers to the exponents. The first argument is a vector containing the DA parameters such as the order and the number of variables, v, and it is the same vector as WRITEM returns. The supplied DA parameters are checked for the compatibility against the current DAINI setup. For all the possible monomials under the current DAINI setup, the corresponding exponents are returned to the third argument. The third argument is an array, and the M-th array element contains the exponents of the M-th monomial, where M is the COSY DA internal number. Each array element is a number (if $v = 1$), or a vector (if $v > 1$) consisting of v components. Supply the length of the array via the second argument.

- **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 $\vec{c}$ 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 $\vec{c}$, and $c(k+1)$ represents the norm of the k-th power part of the DA. The number of returned elements of $\vec{c}$ 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. When order weighted DA is used, it extracts order weighted coefficients. Arguments are the DA, the size of the array (the next argument), and the array containing "linear" coefficients.

- **DACQLC** (ccvvv)
  Extracts coefficients up to second order of a DA. When order weighted DA is used, it extracts order weighted coefficients. Arguments are the DA, 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 $x^t Hx/2 + Lx + c$.

- **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.

- **DANOW** (ccv)
  Computes the order weighted max 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** (vvvvvvv)
  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(: f_2 :)) act on first argument in Floquet variables. Other Arguments: 3 tunes (2π), result.

- **CDNF** (vvvvvvvv)
  Lets $1/(1 - \exp(: f_2 :))$ act on first argument in Floquet variables. Other Arguments: 3 tunes (2π), 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.

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

- **LINV** (cvccv)
  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** (cvvvcc)
  Computes the eigenvalues and eigenvectors of a matrix. Arguments are the matrix A, the real and imaginary parts of eigenvalues, a matrix V containing eigenvectors 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.

- **RECST** (ccv)
  Converts a real or a complex to a string using a Fortran format. Arguments are the real (or complex), 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.
- **IMUNIT** (v)
  Returns the imaginary unit i.
- **LTRUE** (v)
  Returns the logical value true.
- **LFALSE** (v)
  Returns the logical value false.
- **INTPOL** (vc)
  Determines coefficients of Polynomial satisfying $P(\pm 1) = \pm 1$, $P^{(i)}(\pm 1) = 0$, $i = 1, ..., n$. Arguments: coefficient array, n.
- **CLEAR** (v)
  Clears a graphics object.
- **GRMOVE** (cccv)
  Appends one move to a graphics object. Arguments are the three coordinates x, y, z and the graphics object.
- **GRDRAW** (cccv)
  Appends one draw to a graphics object. Arguments are the three coordinates x, y, z and the graphics object.
- **GRDOT** (cccv)
  Appends one move and one dot to a graphics object. Arguments are the three coordinates x, y, z and the graphics object.
- **GRTRI** (cccv)
  Appends a triangle to a graphics object. The triangle is formed by the last two positions and the given point, and updates the current position. Arguments are the three coordinates x, y, z of the newly given point and the graphics object.
- **GRPOLY** (cccv)
  Appends a polynomial curve or surface patch to a graphics object. The curve or surface is specified by an array of DA vectors in one or two variable describing the x, y, z components using three array elements as first argument. The color is specified either by GRCOLR style, namely the color ID number (RE) or a vector (VE) of RGBA values (see GRCOLR), or by the previously set color by GRCOLR style if -1, or by an array of color polynomials describing RGBA using four array elements in the second argument. The third argument describes the independent variable(s) of the position and color polynomials as type RE for the curve case, or VE for the surface case. It is possible to specify the discretization number(s) by using an array for the third argument. In this case, the second component of the array specifies the discretization number(s) corresponding to the independent variable(s). The fourth argument contains the graphics object.

- **GRCURV** (ccccccccv)
  Appends a cubic spline curve to a graphics object. Arguments are the three final coordinates x_f, y_f, z_f, the three components of the initial tangent vector t_{ix}, t_{iy}, t_{iz}, the three components of the final tangent vector t_{fx}, t_{fy}, t_{fz}, and the graphics object.

- **GRCHAR** (cv)
  Adds a string of characters 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 number (RE) or a vector (VE) of RGBA values, and the graphics object. RGBA describes red, green, blue and alpha (opacity), and values are between 0 and 1. When the graphics driver supports alpha, A=1 is opaque (default), and A=0 is transparent and thus invisible. A can be omitted.

color ID	color	R	G	B
1	black (default)	0	0	0
2	blue	0	0.2	1
3	red	1	0	0
4	yellow	1	1	0
5	green	0	1	0
6	yellowish green	0.6	0.9	0.2
7	cyan	0	1	1
8	magenta	1	0	1
9	navy	0	0.2	0.7
10	white	1	1	1

- **GRWDTH** (cv)
  Adds a width change to a graphics object. Arguments are the new width and the graphics object. The default value is 1.

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

- **GRZOOM** (ccccccv)
  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 $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, $z_{\min}$, $z_{\max}$.

- **GREPS** (cv)
  Sets drawing absolute error tolerances for GRPOLY for the approximation of curves or surfaces by line segments or quadrilateral meshes, respectively. If 0, it resets to the default value. Be aware that unreasonably small values may lead to exceedingly large graphics objects. Color error tolerance can also be specified by using a vector instead of a scalar argument, where the second component denotes the desired accuracy in color. Arguments are the absolute tolerance (RE or VE) and the graphics object. The default tolerance is 0.5% of the frame size of the graphics object for the x, y, z part and 0.5% of the full range, i.e. 1, for the color part.

- **GRSTYL** (cv)
  Sets the drawing style. Arguments are the style option and the graphics object. The default option value is 0. If the option is set to 1, the surface by GRPOLY or GRTRI is drawn by wire frame. The default is fill painting.

- **GROUTF** (cc)
  Sets the prefix and the sequence starting number for the graphics output file name used in graphics drivers outputting data to a file. The arguments are the prefix string and the sequence starting number. The default is 'pic' and 1. This is useful to prevent parallel COSY processes from overwriting each other's graphics output.

- **GUISET** (ccc)
  Updates the value of the n-th input field in the given GUI window unit. Arguments are the GUI window unit, the number of the input element to replace, and the new value. Also works for graphics objects.

- **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 2.5 for details.

B Quick Start Guide for COSY INFINITY

This guide is intended to assist new users to quickly get started with COSY INFINITY. The main emphasis is placed on writing a meaningful COSYScript program (with the file extension "`.fox`"), especially for performing Beam Physics computations. Some examples in this guide require to use the Beam Physics macro package cosy.fox.

For the information on how to install and execute COSY INFINITY, refer to the web page http://cosyinfinity.org, and Section 1.5 (page 8) for the installation, and Section 1.7 (page 20) for the execution, and especially Section 1.7.6 (page 22) for executing cosy.fox.

B.1 Basic Structure of a COSYScript Program

B.1.1 Program Segments

A complete COSYScript program consists of a tree-structured arrangement of nested program segments. There are three types of program segments.

MAIN Program

There has to be one main program in a complete COSYScript program. The main program begins at the beginning and ends at the end of the whole program.

Main Program

`BEGIN ;`
...
`END ;`

Procedure Program and Function Program

A COSYScript program can contain many procedures and functions which can be called by the main program and the other procedures or functions. A procedure program and a function program must contain at least one executable statement.

Procedure Program
`PROCEDURE` name { name1 ... } ; ... `ENDPROCEDURE ;`

Function Program
`FUNCTION` name name1 { ... } ; ... `ENDFUNCTION ;`

Note: { } indicates an optional expression.

name: The name of the procedure or the function.

name1 ...: The local name(s) of variable(s) that are passed into the procedure or into the function. These variables are to not be declared inside the procedure or the function.

A call to a procedure program
name { name1 ... } ;

A call to a function program
name (name1 {, ... })

name: The name of the procedure or the function.

name1, ...: The argument(s) that are passed into the procedure or the function.

- The number of arguments in the procedure program or in the function program has to agree with the number of arguments in its calling statements.
- A call to a function program can be made in an arithmetic expression.

Examples

1. A call to the procedure DL.
   `DL .1 ;`
   This is a drift of length .1 m.
2. A call to the function ME.
   `ME(1,2)`
   This is the (x, a) element of the map.

DL, ME are available via cosy.fox; refer to the Beam Physics Manual for DL, ME.

B.1.2 Three Sections inside each Program Segment

Inside each program segment, there are three sections.

1. Declaration of Local Variables

The types of variables are free at the declaration time. There is no distinction among integer, real and double precision numbers. All locally declared variables are visible inside the program segment.

Variable Declaration
`VARIABLE` name exp { exp1 ... } ;

name: The name of the variable to be declared.

exp: The amount of memory to be allocated to the variable.

exp1, ...: In case of an array with indices, it specifies the different dimension.

Examples

1. A real number variable X.
   `VARIABLE X 1 ;`

2. A 5×7 array Y of memory length 100 per array element.
   `VARIABLE Y 100 5 7 ;`

2. Local Procedures and Functions Any local procedures and local functions are coded inside the program segment. Any local program is visible in the segment, as long as a call statement to it is made below the local program.

3. Executable Statements Executable statements are assignment statements, call statements to procedures, flow control statements, input/output statements.

Assignment Statement
variable `:=` expression `;`

variable : The name of a variable or an array element.

expression : A combination of variables and array elements visible in the segment, combined with operands and grouped by parentheses.

Examples

1. An assignment of .5 to a variable Q1.
   `Q1 := .5 ;`

2. An assignment of the summation of the absolute values of (x, a) and (y, b) elements of the map to a variable OBJ.
   `OBJ := ABS(ME(1,2))+ABS(ME(3,4)) ;`

ME is available via cosy.fox; refer to the Beam Physics Manual for ME.

B.2 Input and Output

The basic Input and Output statements are as follows.

READ statement
`READ` unit name `;`

unit: The device unit number. 5 denotes the keyboard.

name: The name of the variable to be input.

WRITE statement
`WRITE` unit name { name1 ... } `;`

unit: The device unit number. 6 denotes the display.

name, name1, ...: The name(s) of the variable(s) or the string(s) to be output.

A PM statement prints the map.

map printing statement
`PM unit ;`

unit: The device unit number. PM is available via cosy.fox; refer to the Beam Physics Manual for PM.

B.3 How to use COSY INFINITY in Beam Physics Computations

There is a COSYScript macro program cosy.fox, which contains many procedures and functions for Beam Physics computations. It forms a portion of a complete COSYScript program. To access those procedures and functions, the user has to include cosy.fox into the user's own COSYScript code. Since cosy.fox starts with the "`BEGIN ;`" statement, the user code has to have the executable code for the main program and the "`END ;`" statement to complete the whole COSYScript program.

To include a COSYScript macro program into the user's code, an include statement has to be placed in the beginning.

Include Statement
`INCLUDE 'name' ;`

name: The name of a previously compiled macro program to be included.

Examples

1. Include the compiled version of cosy.fox.
   `INCLUDE 'COSY' ;`

2. A user's COSYScript code may look as follows.

```
INCLUDE 'COSY' ;
PROCEDURE RUN ;
...
ENDPROCEDURE ;
RUN ;
END ;
```

Tips

- Refer to the Beam Physics Manual for the available procedures and functions in cosy.fox.

B.4 Example: a Sequence of Elements

As a practical example of Beam Physics computations, we set up a sequence of beam elements consisting of a few drifts and a few quadrupoles, and compute a nonlinear transfer map of the sequence. OV, RP, UM, DL, MQ, PM in the example are available via cosy.fox; refer to the Beam Physics Manual. This example program is available as beamdemo_ele.fox at the COSY INFINITY download site.

```
INCLUDE 'COSY' ;
PROCEDURE RUN ;
  OV 5 2 0 ; {order 5, phase space dim 2, # of parameters 0}
  RP 10 4 2 ; {kinetic energy 10MeV, mass 4 amu, charge 2}
  UM ;    {sets map to unity}
  DL .1 ; {drift of length .1 m}
  MQ .2 .1 .05 ; {focusing quad; length .2 m, field .1 T, aperture .05 m}
  DL .1 ;
  MQ .2 -.1 .05 ; {defocusing}
  DL .1 ;
  PM 6 ; {prints map to display}
  ENDPROCEDURE ;
RUN ; END ;
```

The first few lines of the resulting transfer map look like this:

```
  0.7084973    -0.1798231      0.000000      0.000000      0.000000     100000
  0.6952214      1.234984      0.000000      0.000000      0.000000     010000
   0.000000      0.000000      1.234984    -0.1798231      0.000000     001000
   0.000000      0.000000     0.6952214     0.7084973      0.000000     000100
 -0.7552786E-01-0.5173667E-01  0.000000      0.000000      0.000000     300000
  0.2751173     0.1728297      0.000000      0.000000      0.000000     210000
 -0.4105720    -0.2057599      0.000000      0.000000      0.000000     120000
  0.3541071     0.8137949E-01  0.000000      0.000000      0.000000     030000
   0.000000      0.000000     0.5676314E-01-0.5150461E-01  0.000000     201000
```

The different columns correspond to the final coordinates x, a, y, b and t. The lines contain the various expansion coefficients, which are identified by the exponents of the initial condition. For example, the third column, hence the final coordinate y, of the last line is the number `0.5676314E-01`, where the exponents are noted as `201000`, which means xxy. So, the value of the expansion coefficient (y, xxy) is 0.05676314.

Tips

- A comment in COSYScript can be written inside a pair of curly brackets.
  Example
  `{This is a comment in COSY.}`
- Any user executable code for a Beam Physics calculation should start with "`OV`", then "`RP`" (or "`RPP`" or "`RPE`"), then "`UM`". A definition of the beam system consisting of elements like "`DL`", "`DI`", "`MQ`" ... follows afterward.
- demo.fox includes many example calculations with COSY INFINITY. It is a good starting point to refer to demo.fox to find some COSYScript example programs.

The following are typical **tips** for COSY beginners.

- An input COSYScript file name has to have the extension "`.fox`".
- Don't forget to use the delimiter "`;`" at the end of each statement.
- COSYScript expressions are not case sensitive (except for strings treated as STring data type objects).

B.5 Flow Control

Like other computer languages, COSYScript has branching and looping statements. "FIT - ENDFIT structure" is a unique and unusual feature not found in other languages. It enables nonlinear optimization as a part of the syntax of the language.

IF - (ELSEIF) - ENDIF Structure

IF - (ELSEIF) - ENDIF structure

`IF` logical-expression `;`
...
{ `ELSEIF` logical-expression `;`
... }
`ENDIF ;`

Example

1. If the value of X is not zero, assign the multiplicative inverse of X to Y.
```
IF X#0 ; Y:= 1/X ; ENDIF ;
```

WHILE - ENDWHILE Structure

WHILE - ENDWHILE structure

`WHILE` logical-expression `;`
...
`ENDWHILE ;`

Example

1. While the value of N is positive, add N to a variable SUM.
```
SUM := 0 ; READ 5 N ;
WHILE N>0 ; SUM := SUM+N ; READ 5 N ; ENDWHILE ;
```

LOOP - ENDLOOP Structure

LOOP - ENDLOOP structure
`LOOP` name start end { step } ; ... `ENDLOOP` ;

name: The name of the loop counter.

start: The starting value of the counter.

end: The ending value of the counter.

step: The step size of the counter.

Examples

1. Compute 10! and store the result in a variable N.
```
N := 1 ;
LOOP I 1 10 ; N := N*I ; ENDLOOP ;
```

FIT - ENDFIT Structure

FIT - ENDFIT structure
`FIT` name1 { ... } ; ... `ENDFIT` eps max algo o1 { o2 ... } ;

name1 ...: The variables to be fit.

eps: The tolerance.

max: The maximum number of iterations.

algo: The number of optimizing algorithm to be used.

1: The Simplex algorithm.

4: The LMDIF optimizer. Several objective quantities can be specified.

o1 {, o2, ...}: The name(s) of objective quantity (quantities) to be minimized.

Examples

1. See the next example COSYScript program.

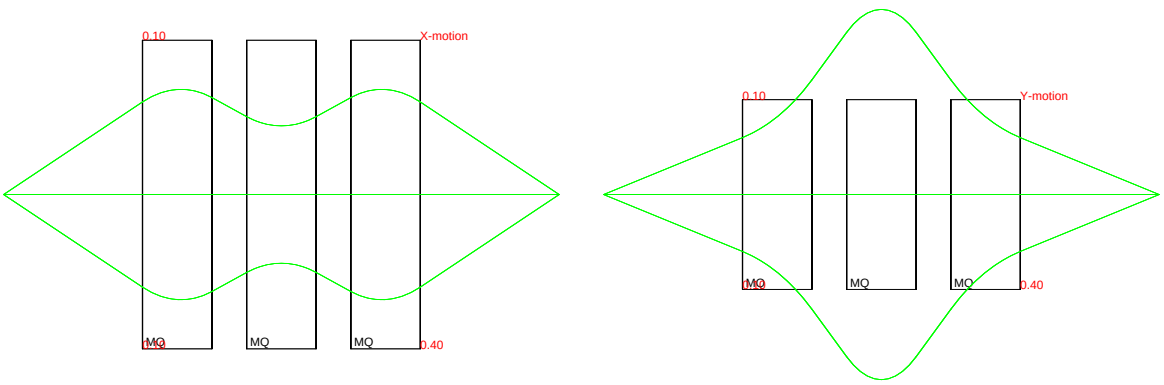
B.6 Example: Fitting a System

As another practical example of Beam Physics computations, we set up a triplet system consisting of three quadrupoles and drifts, and optimize the triplet system to fulfill some conditions, in this case, to form a stigmatic imaging system. The program is set so that we can monitor the process of optimization by

beam trajectories through graphics output. OV, RP, UM, DL, MQ, SB, ER, CR, BP, EP, PG, ME in the example are available via cosy.fox; refer to the Beam Physics Manual. This example program is available as beamdemo_fit.fox at the COSY INFINITY download site.

```
INCLUDE 'COSY' ;
PROCEDURE RUN ;
  VARIABLE Q1 1 ; VARIABLE Q2 1 ; VARIABLE OBJ 1 ;
  PROCEDURE TRIPLET A B ;
    MQ .1  A .05 ; DL .05 ; MQ .1 -B .05 ; DL .05 ; MQ .1  A .05 ;
    ENDPROCEDURE ;
  OV 1 2 0 ; RP 1 1 1 ;
  SB .15 .15 0 .15 .15 0 0 0 0 0 0 ;
    {sets half widths of beam .15 m in x, y and .15 rad in a, b}
  Q1 := .5 ; Q2 := .5 ; {start values of Q1, Q2}
  FIT Q1 Q2 ;
    UM ; CR ; {clears the rays}
    ER 1 3 1 3 1 1 1 1 ; {ensemble of rays, 3 in a, b}
    BP ; {begins a picture}
    DL .2 ; TRIPLET Q1 Q2 ; DL .2 ;
    EP ; {ends the picture}
    PG -1 -2 ; {outputs the x,y pictures to default windows}
    OBJ := ABS(ME(1,2))+ABS(ME(3,4)) ;
      {defines the objective OBJ.
       ME(1,2): map element (x,a),  ME(3,4): map element (y,b)}
    WRITE 6 'Q1, Q2: ' Q1 Q2  'OBJECTIVE: ' OBJ ;
    ENDFIT 1E-5 1000 1 OBJ ;
      {fits OBJ by Simplex algorithm. This is point-to-point for both x, y}
  PG -12 -12 ;
    {output final pictures to PDF files pic001.pdf and pic002.pdf}
  ENDPROCEDURE ;
RUN ; END ;
```

The following final x, y pictures are created in PDF files pic001.pdf and pic002.pdf.



Tip: Refer to Section 5.2 (page 39) for information on the device unit numbers for various graphics drivers.

Index

Z