

cosin

scientific software

Gipser + Hofmann, Ingenieure, Partnerschaft

Agnes-Pockels-Bogen 1
80992 München
GERMANY

info@cosin.eu
www.cosin.eu

cosin/io
Cosin Data File Format
Documentation and User's Guide

Contents

1	Specification of Input and Output Files	1
2	Syntax of Input Files	1
3	Arithmetic Expressions	7
4	Look-up Tables, Splines, and Polynomials	18
5	Specification of Control Variables (Sources & Sinks)	21
6	Plot-File Formats	24

Preface

cosin/io is a program library, which allows generating easy-to-use simulation environments under Windows and Unix, together with the definition of user-friendly input/output file formats. This user's guide describes these file formats, and shows how to work with simulation models in *cosin/io* environments. All product or brand names mentioned here are trademarks or registered trademarks of their respective holders.

cosin/io supports

- the input of parameterized simulation and model data,
- the definition of input and control variables for the simulation model,
- the generation of online-plots and online-animations, and
- the output of plot data in several formats.

cosin/io consists of the following program packages:

- AR** routines for interpreting and calculating arithmetic expressions
- BF** routines for *cosin/io* file handling
- BP** online plot functions
- EV** 'environment', functions for the description of location- and time-dependent environment attributes for vehicle dynamics simulation (road profiles, friction coefficients, wind velocities, etc.)
- IO** routines for reading and interpreting structured and parameterized input files, interactive user input, and screen output
- PO** routines for output of plot data in different formats
- SP** routines for defining and calculating splines and polynomials for use in several other packages
- SS** 'Sources&Sinks', routines for defining input- and output signals for simulation models

With *cosin/io*, several functions for the execution of simulations are available. Depending on the model at hand, the library can also be used in parts only. The documentation focuses on the following subjects:

- the specification and format of input and output files,
- the syntax of arithmetic expressions used in input files,
- the syntax of spline definitions,
- the definition and syntax of control variables,
- plot file formats.

1 Specification of Input and Output Files

cfid-file

There are two ways of specification of input and output file-names in *cosin/io*-driven simulation programs:

- either the program asks for the file-names during execution, or
- the program reads the file-names in a special file, the so-called **cosin/io-file definition file** (cfid-file).

Usually, the cfd-file is edited before execution of the simulation program, or it is automatically generated when working in a workbench menu environment. In the first case, the user has the opportunity to 'manually' specify the files to be used. In the second case, the user specifies these files by clicking appropriate fields in the menu. The cfd-file itself is a *cosin/io* input file. Hence, it has to comply with few syntax rules as specified in the subsequent chapter.

file-identifier

If the cfd-file is used, there are no references to 'physical' file-names in the simulation program. Instead of that, logical file-names, the so-called file identifiers, are used. In the cfd-file, the physical file-names are assigned to the logical file-names. Example: when using

```
ipar = mypar/input.dat  
oplot = myplot/plot.dat
```

the files `mypar/input.dat` and `myplot/plot.dat` are assigned to the file- identifiers `ipar` and `oplot`. File-identifier names usually are hard-coded in the simulation program. Usually, file-identifiers for input files start with letter **i**, file-identifiers for output files with letter **o**.

*** as file-name**

If the 'file-name' `*` is assigned to a file-identifier, the cfd-file itself is referenced. In this way, all input data of small projects may be collected in a single file, even if they logically belong to different files.

\ equals /

In the cfd-file, paths in Windows file-names may be specified using either `\` or `/`. Therefore, it is possible to use the same cfd-file in Windows and Unix environments.

2 Syntax of Input Files

**ASCII-files line name
length \leq 256**

cosin/io-input files are ASCII-files that can be generated and modified with any given editor. The line length is limited to 256 characters.

* comment line ..
! comment

cosin/io-input files may comprise any number of comment lines. These are characterized by an asterisk (*) as the first character in line. Comment lines as well as empty lines are ignored. Within a line, characters following an exclamation mark are also interpreted as comment. This rule doesn't apply if the exclamation mark is part of a string, which itself is enclosed in quotation marks (see below for further information).

;
lines and records

Besides the 'physical' partitioning into lines, *cosin/io*-input files can be 'logically' partitioned into different **records**. After eliminating any comments, these records are separated either by line spacing ('Return') or by a semicolon. Hence, the file section

```
x=30; y=10 ! first line, consisting of two records  
z=40 ! second line, consisting of one record
```

consists of three records: 'x=30', 'y=10', and 'z=40'. Usually, but not necessarily, each record starts in a new line

up to 2000 records

Since *cosin/io*-input files are completely stored in memory before interpretation, the number of records is limited. For each input file, in the standard version of *cosin/io*, a maximum number of 2000 records is predefined. This will be sufficient in most cases.

**arbitrary sequence of
records**

Records within a data block may appear in any arbitrary sequence. Hence, the two lines

```
y=10; z=40; x=30  
z=40; y=10; x=30
```

as well as the group of lines

```
x=30  
y=10  
z=40
```

are equivalent.

name = value
or
name value

cosin/io-input files are most frequently used for assigning numerical values to variables. If the simulation program is searching for a variable *x*, say, the corresponding value can be specified in the *cosin/io*-input file by using the record *x=30*. The variable name *x* may be preceded by an arbitrary number of spaces. Furthermore, there may be also an arbitrary number of spaces around the equality sign. The equality sign itself is optional. If it is not used, there must be at least one space between the name and the value of the variable. For each scalar variable, a new record is required.

name = text
or
name = 'text1 text2'

Instead of numerical values, strings as well may be assigned to variables. They are specified either with or without quotation marks. As for numerical values, the use of an equality sign is optional. When using spaces, quotation marks, commas, semicolons, or exclamation marks, the string has to be enclosed by quotation marks, see below.

within strings:
'x'abc'y'
equals
x'abc'y'

Within a string, arbitrary characters may occur, if the string is enclosed by quotation marks. A quotation mark within a string is specified by double quotation marks. The string enclosing quotation marks are not part of the variable.

for variable names:
abc equals ABC

Variable names are **not case sensitive**. A variable name may not comprise any spaces, exclamation marks, commas or semicolons. This has to be ensured by the simulation program.

vectors
matrices

According to the specification in the simulation program, vectors and matrices may also be defined by a single value assignment. In this case, the component values are allowed to extend over several records and lines, respectively. The dimensions of the vectors and matrices are only limited by the memory reserved in the simulation program. Matrices are read row by row. For example, the 3x4-matrix

$$A = \begin{bmatrix} 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \\ 12 & 13 & 14 & 15 \end{bmatrix}$$

is specified by the records

```
A = 4 5 6 7 ! first matrix line
```

```
8 9 10 11 12 13 14 15 ! second and third matrix line
```

The components have to be delimited by at least one separating sign. Valid separators are spaces, commas, semicolons, and line separators ('Return'). If the number of given numerical values is less than the number of components, the value 0 is assigned to the missing components.

`name = 3.0e+1`

The numerical value of a scalar variable or array component may be specified either with or without a decimal point, with or without a sign, and with or without an exponent. The following records are all equivalent:

```
x=30. , x=+30. , x=3.0e1 , x=300E-1, etc.
```

`name = 3*a + exp(b)`

Instead of a numerical value for a variable or a variable component, an arbitrary **arithmetic expression** may be specified. The syntax of arithmetic expressions is described in the next section. If the expression comprises spaces, quotation marks, commas, semicolons or exclamation marks, it has to be enclosed by quotation marks.

`name = ?`

Instead of specifying a numerical value for a variable or a variable component, a question mark can be used. In this case, the user will be asked for the current value, while *cosin/io* is reading the file.

up to 100 data blocks

An input file may be sub-divided up into 100 **data blocks**, depending on the definition in the simulation program.

\$ data block
\$\$ sub-block
\$\$\$ sub-sub-block

A new data block begins with one, two or three \$-signs (without spaces in between), followed by a string, which denotes the name of the data block. Spaces may precede the first \$-sign and follow the last \$-sign. Data block names are **not case-sensitive**. A data block may be **optionally** closed by the same number of \$-signs, which were used when it was opened. In the closing line, no data name is allowed.

data block level

The number of \$-signs determines the **level** of the data block. Data blocks are arranged in a tree structure corresponding to their level. The beginning of a new data block with a level not greater than the level of the preceding block, at the same time marks the end of the preceding block. Different data blocks may comprise identical variable names. If a variable name occurs more than once in a data block, the value of first occurrence is used. The sequence of data blocks of the same level is arbitrary.

tree structure

The tree structure, which allows for the hierarchical ordering in subblocks, is determined by the simulation program and shall be documented there. In case the input file is divided into several data blocks, each variable has to be specified in the 'correct' block.

up to 100 parameters within arithmetic expressions

In arithmetic expressions, up to 100 **parameters** (= variables, see below) may be used. The current values of the parameters are defined within one of the data blocks \$parameters or \$more_parameters. In doing so, a parameter may be replaced by an arithmetic expression, which itself contains parameters. Clearly, the latter must be already defined at the time of evaluation of the first. Example:

\$parameters

```
$parameters  
a=3; b=2*a; c=3*a*b  
$data  
x=exp(c)
```

The parameter definition is only valid during loading of the current input file.

#meta-command

If the first non-space character of a record is a hash character (#), the remaining part of the record is interpreted as a meta-command.

`#echo`

The meta-command `echo` initializes the output of a so-called echo-file. The echo-file shows all variables together with their values, eventually after evaluation of arithmetic expressions.

If the current input file was specified by a file-identifier, the name of the echo-file is built according to the following scheme:

`fxFI.eco`

(*x* is a numbering of all echo-files, FI stands for the file-identifier).

The echo files are numbered in the sequence in which they are

generated. Example: An echo-file `f2IPAR.eco` is generated in the working directory, if this is ordered by the file with identifier `ipar`.

The echo file's name further indicates that it is the second echo-file during execution of the program.

If the input file-name is given (instead of the file-identifier), the

name of the echo-file is built from the file-name by replacing the

given file extension by the extension `.eco`. Example: An input file `myparm/input.dat` might generate the echo-file

`myparm/input.eco`.

`#quiet`

With the meta-command `quiet`, the echo output is suppressed or stopped, respectively. This is the default setting.

3 Arithmetic Expressions

operators operations

In *cosin/io*-files, **arithmetic expressions** are represented by character strings, which consist of

- number constants,
- mathematical constants,
- variables of different type and origin,
- random numbers,
- arithmetic operations,
- boolean operations,
- comparison operations,
- mathematical functions,
- special functions, and
- bracket expressions.

cosin/io interprets arithmetic expressions according to the well-known arithmetic operator precedence rules. Additional rules, which are not commonly used, are discussed below.

In *cosin/io*-input files, arithmetic expressions have to be enclosed by quotation marks, if spaces, quotation marks, commas, semicolons, or exclamation marks are part of the arithmetic expression itself (see above).

constants

In *cosin/io*-input files, the rules applied for number constants are identical to those for variable values. A number constant may be specified either with or without a decimal point, with or without a sign, and with or without an exponent. For example, the following representations of the constant 30 are equivalent: 30. , 30.0 , +30. , 3.0e1 , 300E-1, etc.. Within number constants, no blank spaces are allowed.

variables

In arithmetic expressions, **variables** (parameters) may be used instead of number constants. The variables and their respective values are stored in the *cosin/io*-input file within one of the data blocks *\$parameters* or *\$more_parameters*, see above. To those variables which are not defined, the value 0 is assigned. In addition, when using *cosin/ss*, all **input signals** are automatically provided as variables. In all *cosin* applications that are generating *cosin/ip* output, also the **plot signals** are provided as variables.

\string

If a string, preceded by a back-slash (`\`), is used in an arithmetic expression, the value of the first variable, the name of which contains this string, will be used. The value 0 is assigned if none of the variable names contains the string.

abc_123 equals ABC_123

The lengths of variable names are limited to 31 characters. Like in C or Matlab programming language, the only permitted characters are letters, numerals and the underscore (`_`). It is not recommended to use a digit as first character. Variable names are **not case-sensitive**.

pi

e

r2d

d2r

%pi

%e

%d2r

%r2d

The four specific variables

- `pi` = π = 3.141592654...
- `e` = 2.718281828...
- `d2r` = $\pi/180$
- `r2d` = $180/\pi$

are predefined and may be used in each arithmetic expression. If used without %-sign, these constants are subject to redefinition: the user may assign different values in the data blocks *\$parameters* and *\$more_parameters*. By using the character %, redefinition can be avoided.

rnd

rndn

%rnd

%rndn

`rnd` and `rndn` are predefined variables, which generate a **random number** after they are called:

- `rnd` generates an equally distributed number in the interval $[0,1]$,
- `rndn` generates a normally distributed number with mean value 0 and standard deviation 1.

Again, `rnd` and `rndn` are 'redefinable', see above.

?

Instead of a number, a question mark may be used. In this case, the user is asked for the corresponding value during interpretation time of the arithmetic expression.

"

Any double quotation mark is replaced by the result of the preceding arithmetic expression. Example:

`a=rnd; b="^2`

assigns an equally distributed number to the variable a. Then, a is squared and the result assigned to b.

+

The following arithmetic operations are permitted:

-

*

/

^

**

- +
- -
- *
- /
- ^(or**)

The minus sign may be also used as unary operator: $-x+y$.

In *cosin/io*, the well-known precedence rules for arithmetic expressions are observed. Operations on the same level are evaluated from the left to the right hand side, that is $a/b/c$ equals $(a/b)/c$.

.and.

.or.

.nand.

.exor.

.not.

&

|

§

\$

!

~

Arithmetic expressions may also contain **logical operations**. The value 'false' is assigned, if the absolute value of a variable is less equal 10^{-10} . Otherwise the value 'true' is assigned. All logical operations result in a value 0 or 1. Permitted operations are:

- .and. (or &)
- .or. (or |)
- .nand. (or §)
- .exor. (or \$)
- .not. (or ! , ~)

(note the leading and trailing dots. These are needed to distinguish the operators from variables). The precedence sequence for logical operations is as follows: not, and, nand, exor, or. Operations on the same level are evaluated from the left to the right hand side.

`a < b + c` equals

`a < (b + c)`

`=`

`<`

`>`

`<=`

`>=`

`<>`

`eps=`

Arithmetic operations have priority over logical operations.

The following comparison operators are permitted:

- `=` (or `==`)
- `eps=`
- `<`
- `>`
- `<=`
- `>=`
- `<>` (or `!=` or `~=`)

Comparison operations result in the value 1, in case they are fulfilled, otherwise they result in the value 0. As in all programming languages, two variables are equal only if they exactly correspond. Therefore, another operator `eps=` ('almost equal') is available: `x eps= y` is fulfilled, if $|x-y| \leq 10^{-10}$ (i.e., if they are **numerically** nearly equivalent).

All comparison operations are of the same priority. They are evaluated from the left to the right hand side.

`a < b .and. b < c`

equals

`(a < b) .and. (b < c)`

Comparison operators have priority over logical operators. Hence, parentheses in a sequence of logically combined comparison operations are often redundant.

sin(.)
cos(.)
tan(.)
cot(.)
asin(.)
acos(.)
atan(.)
acot(.)
atan2(.)
exp(.)
log(.)
ln(.)
sqrt(.)
sinh(.)
cosh(.)
abs(.)
min(.)
max(.)

The following **mathematical functions** with their respective valid argument ranges are permitted:

sin(x)
cos(x)
tan(x)
cot(x)
asin(x)
acos(x)
atan(x)
acot(x)
atan2(x,y)
exp(x)
log(x)
ln(x)
sqrt(x)
sinh(x)
cosh(x)
abs(x)
min(x,y)
max(x,y)

As usual, arguments of angular functions are specified in radians. Arguments of functions in turn may be arithmetic expressions. For example, a function value can be used as an argument of another function.

In case of **execution errors** (e.g. if a function is called with an illegal argument), the corresponding error handling routine of the compiler is called, which in most cases will terminate the program run.

int(.)
der(.)
pt1(.)
pid(.)

The following **dynamic functions** are permitted:

- `int(x)` integrates expression `x` with respect to time `t`, if `t` appears as variable in the variable list
- `der(x)` differentiates expression `x` with respect to time `t`, if `t` appears as variable in the variable list
- `pt1(x,T)` low-pass filters expression `x` with respect to time `t`, if `t` appears as variable in the variable list:
 - `T` is the filter time constant in [s]
- `pid(ya,yn,Kp,Ti,Td,umin,umax)` is a PID feed-back controller, if `t` appears as variable in the variable list.

Parameters of `pid(·)`:

- `ya` is the actual plant output,
- `yn` is the nominal plant output,
- `Kp` is the proportional gain,
- `Ti = Kp/Ki` defines the integral gain,
- `Td = Kd/Kp` defines the differential gain,
- `umin` and `umax` are the controller output lower and upper bounds. `pid(·)` is an implementation of the general PID controller law:

$$\bar{u} = K_p \left((y_n - y_a) + \frac{1}{T_i} \int_{t_0}^t (y_n - y_a) dt + T_d \frac{d}{dt} (y_n - y_a) \right)$$
$$u = \max(u_{min}, \min(u_{max}, \bar{u}))$$

pconst(.)
linear(.)
spline(.)
poly(.)
bilin(.)
bicub(.)

The following **functions for table data** are provided:

- `pconst(file,datablock,x)` evaluates a piecewise constant function of one independent variable x (cf. chapter 1.4). The data points are defined in 'data-block' that is looked for in 'file'. If 'file' equals '*', the actual input file is used, where the function definition appears
- `linear(file,datablock,x)` evaluates a piecewise linear function of one independent variable x (cf. chapter 1.4). The data points are defined in 'data-block' that is looked for in 'file'. If 'file' equals '*', the actual input file is used, where the function definition appears
- `spline(file,datablock,x)` evaluates a cubic spline function of one independent variable x (cf. chapter 1.4). The data points are defined in 'data-block' that is looked for in 'file'. If 'file' equals '*', the actual input file is used, where the function definition appears
- `poly(file,datablock,x)` evaluates a polynomial of one independent variable x (cf. chapter 1.4). The coefficients are defined in 'data-block' that is looked for in 'file'. If 'file' equals '*', the actual input file is used, where the function definition appears
- `bilin(file,datablock,x,y)` evaluates a piecewise bilinear function of two independent variables x and y (cf. chapter 1.4). The data points are defined in 'data-block' that is looked for in 'file'. If 'file' equals '*', the actual input file is used, where the function definition appears
- `bicub(file,datablock,x,y)` evaluates a piecewise bicubic function of two independent variables x and y (cf. chapter 1.4). The data points are defined in 'data-block' that is looked for in 'file'. If 'file' equals '*', the actual input file is used, where the function definition appears

For all functions in this group, instead of the pair 'file, data-block' a '**raw-data-file-name**' can be entered alternatively. Such a file-name is marked with a '@' preceding the first character. Raw-data files are not subdivided into data-blocks. They consist only of lines with an appropriate number of numerical data. Lines with '*' or '!' as first non-blank character are ignored. Raw-data files can have a nearly arbitrary number of lines, limited only by the size of the internally used memory. As an example,

`linear(@mydata.dat,x)` will evaluate such a data file.

sweep(.)
swlin(.)
frequ(.)
frlin(.)

These functions are used to generate and analyze a swept sine wave.

- `sweep(fstart,ffinal,T,t)` computes a sine wave with varying frequency and amplitude 1. Frequency will be varied in such a way that the cycle length is reduced or increased by a constant factor per cycle ('logarithmic sweep', the preferred and most natural kind of sweep):
 - `fstart` is the initial frequency in Hz,
 - `ffinal` is the final frequency in Hz,
 - `T` is the duration of frequency variation,
 - `t` is the actual time (or independent argument)
- `swlin(fstart,ffinal,T,t)` is parameter compatible to `sweep`, but will vary the frequency at a constant rate ('linear sweep')
- `frequ(fstart,ffinal,T,t)` is parameter compatible to `sweep`, and computes the actual frequency of the logarithmic sweep
- `frlin(fstart,ffinal,T,t)` is parameter compatible to `swlin`, and computes the actual frequency of the linear sweep.

file(.)
meas(.)

These functions are used to read a signal from a data file. The file must be given in one of the valid plot-file formats described below. The actual format will be determined automatically.

- `file(file,signalx,signaly,x)` has the following parameters:
 - `file` is the name of the data file (**not** enclosed in quotes). The file-name must not contain any blank spaces. If a period ('.') as place-holder is used, the file-name is determined by using the file-identifier `isrcf`,
 - `signalx` is the name of the plot-signal, to be defined in the data file, which determines the values of the independent variable. The signal name must not contain blank spaces. To define a blank space, use a period ('.') at the respective position in the string, If `signalx` only consists of a single period, the first signal in the data file will be used,
 - `signaly` is the name of the plot-signal which determines the values of the dependent variable, following the same syntax rules as for `signalx`.
 - `x` is the actual value of the independent variable.
- `meas(signal,x)` will read the signals in the file defined by file-identifier `imeas`. The signal of the independent variable is taken to be the **first** signal in the file, which is usually time. `meas(.)` uses the following parameters:
 - `signal` is the name of the plot-signal, to be defined in the data file, which determines the values of the dependent variable. The signal name must not contain blank spaces. To generate a blank space in the signal name, use a period ('.') at the respective position in the string,
 - `x` is the actual value of the independent variable.

Note: `meas(signal,x)` is equivalent to `file(>imeas,.,signal,x)`.

inival(.)
dur(.)
mean(.)
rms(.)
mins(.)
maxs(.)
trig(.)

These functions extract certain signal properties. In all functions, `file` is the name of the data file (**not** enclosed in quotes) which holds the signal. The file-name must not contain any blank spaces. `signal` is the name of the plot-signal, to be defined in the data file, from which information is to be extracted. The signal name must not contain blank spaces. To define a blank space, use a period ('.') at the respective position in the string. If `signal` only consists of a single period, the first signal in the data file will be used.

- `inival(file,signal)` determines the first value of `signal`,
- `dur(file)` determines the difference between last and first value of the independent signal 'time' in `file`,
- `mean(file,signal)` determines the mean value of `signal`,
- `rms(file,signal)` determines the RMS value of `signal`,
- `mins(file,signal)` determines the minimum value of `signal`,
- `maxs(file,signal)` determines the maximum value of `signal`,
- `trig(file,signal)` determines the first time at which the value of `signal` changes significantly.

param(.)

This function returns the value of a single data item in a *cosin/io*-compatible data file.

- `param(file,db,name)` has the following parameters:
 - `file` is the name of the *cosin/io*-compatible data file (not enclosed in quotes). The file-name must not contain any blank spaces,
 - `db` is the name of the data block in which the variable is to be searched, and
 - `name` is the name of the data item, the value of which is to be returned.

slider(.)
dslide(.)

These functions provide **direct user interaction** with a simulation program. They return the position value of a 'slider', which appears if a cosin/gl window had been opened by the program. By moving one of up to 10 sliders with the left mouse button, its value will vary between definable lower and upper bounds. If slider was called, the values will continuously vary, whereas dslide will return discrete (integer) values.

- slider(label, smin, sinit, smax) has the following parameters:
 - label is a character string and will be displayed in the graphics window, together with the slider. It must not contain blank spaces. To generate a blank space in the display, use a period ('.') at the respective position in the string,
 - smin is the minimum slider value, which will be returned if slider position is left-most,
 - sinit is the initial slider value,
 - smax is the maximum slider value, which will be returned if slider position is right-most.
- dslide(name, smin, sinit, smax) will return a correctly rounded integer value between smin and smax. It has the following parameters:
 - label is a character string and will be displayed in the graphics window, together with the slider. It must not contain blank spaces. To generate a blank space in the display, use a period ('.') at the respective position in the string,
 - smin is the minimum slider value, the rounded value of which will be returned if slider position is left-most,
 - sinit is the initial slider value,
 - smax is the maximum slider value, the rounded value of which will be returned if slider position is right-most.

4 Look-up Tables, Splines, and Polynomials

function types

cosin/io comprises an efficient and easy-to-use package for **look-up tables, spline interpolation, and polynomial functions**. At present,

- piecewise linear functions of one independent variable,
- cubic splines of one independent variable with natural or periodic boundary conditions,
- general polynomials of one independent variable,
- parametric cubic splines for two-dimensional curves,
- piecewise bilinear functions of two independent variables, and
- piecewise bicubic functions of two independent variables are available, together with special interactive browsing tools.

A specialty of *cosin/io* splines is the fact that they can be supplied with an arbitrary number of '**kinks**', such that only continuity of order zero is given. In addition to the function value, the simulation program may also order the evaluation of the first and second derivatives.

If either of the above function types is used by a simulation program, the corresponding data are read from a *cosin/io* input file. The name of that input file either is specified by the simulation program, or by the corresponding cfd-file, see description **on page 1** for further information. Function data are listed within data blocks. The name of the data block is determined by the simulation program and shall be documented there.

table data for one independent variable

The *cosin/io*-spline-routines allow for an arbitrary number N of data points. In particular, the special cases $N = 0, 1, 2$ are valid, too. If no data points are specified ($N = 0$), the value of the function equals 0 by definition. A single data point ($N = 1$) defines a constant function, two data points ($N = 2$) a straight line.

Furthermore, the *cosin/io* spline routines allow **extrapolation**. In this case, a control variable is given as a measure for the distance between the argument and the specified data points.

The maximum number of data points N is 2000 when using *cosin/io* data files, and nearly unlimited if using raw-data files. The simulation program should generate an error message, if too many data points are tried to read.

Within a record of the data block, the N number pairs, which define the spline, are separated by commas. Of course, with each number pair a new record can be used. In the latter case no commas are necessary. Example: the following data blocks define the same spline:

```
$my_spline_1
```

```
3 4, 4 7, 6 9, 10 20
```

```
12 30, 13 14, 16 12
```

```
$my_spline_2
```

```
3 4
```

```
4 7
```

```
6 9
```

```
10 20
```

```
12 30
```

```
13 14
```

```
16 12
```

```
$my_spline_3
```

```
3 4; 4 7; 6 9; 10 20; 12 30; 13 14; 16 12
```

```
$my_spline_4
```

```
3 4; 4 7, 6 9
```

```
10 20
```

```
12 30
```

```
13 14; 16 12
```

The data points may also be defined by **arithmetic expressions**. When using spaces, quotation marks, commas, semicolons or exclamation marks, the expression has to be enclosed by quotation marks.

In order to allow for a 'kink' at a certain data point, the point has to be preceded by the character '^'. If the data point is defined by a quoted arithmetic expression, the character must precede the opening quotation mark. Example: The spline with data

```
$my_spline
```

table data for parametric curves with one independent variable

Table data for parametric curves with one independent variable $(x(s), y(s))$ are similar to those for single functions as described above. Only difference: these tables have three instead of only two columns. That is, data points consist of three instead of two values. The first value corresponds to the independent variable s of a point of the curve, the second value to the x -coordinate of that point, and the third to the y -coordinate.

polynomial data for one independent variable

If a polynomial takes the form $y = p(x) = a_0 + a_1x + a_2x^2 + \dots + a_nx^n$, the corresponding polynomial data are defined by the vector of coefficients $a_0 a_1 a_2 \dots a_n$. Of course, some or all components of that vector may be defined by arithmetic expressions, and all syntax rules for vectorvalued data apply, as described above. The degree n of the polynomial is automatically recognized by the number of coefficients found in the data block.

table data for two independent variables

Table data for two independent variables comprise

- a vector of values for the first independent variable $x_i, i = 1, \dots, n$
- a vector of values for the second independent variable $y_k, k = 1, \dots, m$
- an array of values of the dependent variable $z_{ik}, i = 1, \dots, n, k = 1, \dots, m$.

These values are arranged as follows:

- the first row in the table has n values $x_i, i = 1, \dots, n$
- row 2 to row $m + 1$ each has $n + 1$ values. The first one is y_k , followed by values $z_{ik}, i = 1, \dots, n$.

That is, the table looks as follows:

	x_1	x_2	\cdots	x_n
y_1	z_{11}	z_{21}	\cdots	z_{n1}
y_2	z_{12}	z_{22}	\cdots	z_{n2}
\vdots	\vdots	\vdots		\vdots
y_m	z_{1m}	z_{2m}	\cdots	z_{nm}

Of course, all syntax rules apply for entering that table in the data block, with the following amendment: the first row has one element less than all following rows.

5 Specification of Control Variables (Sources & Sinks)

`$sources`

In general, simulation programs which run in a *cosin/io*-environment use an input file with the file-identifier `isim`. In this input file, the time variant input variables of the simulation model are usually defined within the data block `$sources`. The data block is interpreted and evaluated by the *cosin/io*-package `SS (sources & sinks)`.

input variables parameters

The entries in data block `$sources` usually assign values to variables. The assignment structure is explained in section 1.2. The variables are either **input variables** of the simulation model ('`sources`') or dynamically defined **parameters** for further use in arithmetic expressions within the current data block.

commands

Assignments for input variables are called **commands**. Example: by using the command

```
steering_input = 30*sin(2*pi*f*t)
```

an input variable '`steering_input`' is calculated from the two given variables `f` and `t`. The names of the input variables are defined by the simulation program and shall be documented there.

Names are **not case sensitive**. Input variables may be defined in arbitrary succession. Nevertheless, the chosen succession may have side effects (e.g. by using contradicting commands, see below).

output variables

variables in commands may represent 16

- other input variables,
- dynamic parameters,
- parameters, defined within the data block `$parameters`, or
- **output variables ('sinks')**.

Output variables are defined and written by the simulation program. They should be documented there. Often, time `t` is given as an output variable, for example.

If a variable, which is used in an arithmetic expression, is neither an input variable, nor an output variable, nor a previously defined parameter, an error message is printed. Depending on the simulation program, the execution will be terminated.

condition: command

The execution of commands may depend on certain conditions, which precede the command, separated by a colon. Conditions are arithmetic expressions (mostly logically operated comparisons). The command is only executed, if the condition is fulfilled, that is the corresponding arithmetic expression equals a non-zero value.

Example: by using the conditional command

```
t >= 3 : steering_input = 30*sin( 2*pi*f*(t-3))
```

a steering input of sinusoidal shape is started after 3 seconds of simulation. The same effect can be achieved by the commands

```
tr = t-3 tr: steering_input = 30*sin( 2*pi*f*tr )
```

active commands

A command is called **active**, if its corresponding condition is fulfilled. Commands without any condition or with an empty condition (see below) are always active. The activation or deactivation of commands is reported on the screen.

```
cond1: com1
com2
: com3
com4
```

A condition is not only valid for the immediately following command, but also for all subsequent commands until a new condition is defined. Besides, the empty condition ':' is also allowed, which is always defined as true. With the empty condition, a preceding condition can be 'switched off'. Example: by using

```
t >= 3 : steering_input = 10*(t-3)
gas_pedal = 0.5
t >= 10 : clutch_pedal = 1
: brake_pedal = 0 ; shift = 4
```

a maneuver for 'steering_input' and 'gas_pedal' is triggered after 3 seconds. Furthermore, after 10 seconds, a maneuver for 'clutch_pedal' is triggered. During this time, the variables 'brake_pedal' and 'shift' take the values 0 and 4, respectively.

contradicting commands

If more than one value is tried to be assigned by several commands, then the command, which appeared last, is valid. Example: by using

```
steering_input = 100*t
t >= 1 : steering_input = 100
```

the input variable 'steering_input' is linearly increased up to the value 100. Afterwards, it is kept constant. By the way, the same effect can be achieved by the command

```
steering_input = max(100*t,100)
```

missing commands

To input variables which are not defined by an active command, the following rule applies: if there has been no active command yet, the value 0 is assigned. Otherwise, the last value assigned by the last active command is kept.

vector valued input variables

The simulation program may also ask for the specification of vector valued input variables. In this case, the components have to be defined one after another. The component index is added to the variable name, enclosed in brackets. Example:

```
r[1] = 0 ; r[2] = 2*t ; r[3] = sin(3*t)
```

defines an input vector $r = [0 \ 2t \ \sin 3t]^T$.

signals

Sometimes commands are not definitions of input variables, but **signals**. These signals are sent to the simulation program in order to control the simulation process. For example, the output of plot variables, or online-animation can be switched on or off, etc.. Signals are keywords, which are defined by the simulation program. They should be documented there.

The signal names are **not case sensitive**.

stop

Usually, the predefined signal stop is defined by the simulation program. Stop terminates the simulation. Prior to that, the remaining output data are written to the output files. Then, all files are closed. Example: the command

```
roll_angle >= 30: stop
```

terminates the simulation, if the output variable roll_angle is greater or equal 30.

msg>text

The predefined signal msg>, followed by an arbitrary text, will pop up a message window and display the text in this window.

log>text

The predefined signal log>, followed by an arbitrary text, will put out this text in the log-file, provided the log level currently is greater or equal to 2.

save>file

The predefined signal save>, followed by an arbitrary file name, will save the current values of all source signals in an ASCII-file with name as specified.

6 Plot-File Formats

`plotfile_format`

In the *cosin/io*-input file with file-identifier `isim`, the variable `plotfile_format` can be used for the specification of a plotfile format, if need be. During simulation, the output variables are written to the corresponding plotfile.

In general, this data item can be found in data block `$simulation_data`. Valid values are:

- `excel`
- `gnuplot`
- `matlab`
- `matrix`

Output data are written in the specified format for subsequent evaluation by the respective program. All programs allow an interactive graphical representation of the stored output variables. The simulation program determines which output variables to select. The way of selection should be documented there. Because of its simplicity and versatility, the format `matlab` is recommended.

`excel`

Using the selection `excel`, an ASCII-file is generated, which can be easily loaded in Microsoft **Excel**. In the first line of the table, the names of the output variables are stored.

`gnuplot`

By selecting `gnuplot`, an ASCII-file is generated, which can be evaluated by the GNU program **gnuplot**. Gnuplot is released under the terms of the GPL and available for Windows, Linux, and Unix.

matlab

By selecting `matlab`, output variables are stored as a matrix to be read in in **Matlab**. Matlab allows for comment lines, starting with a percent sign. Plot labels are stored in the file-header, one line per label, using this comment line syntax. In the data section of the file, for each output step a new line is begun. Hence, the columns correspond to 'output channels'. Usually, the time steps are stored in the first column.

Example: An output signal in the file `myplot/result.mtl` in `matlab` format can be loaded and plotted in **Matlab**, using the commands

```
load myplot/result.mtl
plot(result(1,:),result(5,:))
```

(a Matlab diagram will be created, showing the fifth output variable vs. time).

If you prefer to automatically label your Matlab-plot, you will have to write a simple M-script using Matlab-commands `fprintf(.)`, `fscanf(.)`, etc., to read and interpret the comment lines. Such a script is contained in the *cosin/mbs* download. The selection `matlab` is the **default** plot format.

matrix

Output variables are stored as a matrix without any labels. For each output step, a new line is created. As opposed to format `matlab`, plot labels do **not** appear in this data format.