cosin
scientific software

cosin scientific software AG

Hans-Stützle-Str. 20
81249 München
GERMANY

info@cosin.eu
www.cosin.eu

cosin/io

Cosin Data File Format

Documentation and User's Guide

Contents

Document Revision: 2021-2-r24053

Preface

cosin/io is a generic data file format used in all cosin applications. This user's guide describes:

- its syntax,
- the syntax of arithmetic expressions used therein,
- and the syntax of table data and spline definitions used therein.

1 Syntax of cosin/io Data Files

ASCII file: line length $\leq$ 256

cosin/io-formatted input files are ASCII files that can be generated and modified with any ASCII editor. For efficiency reasons, the line length is limited to 256 characters.

*** comment line**
.. ! comment

cosin/io-formatted 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**-formatted 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 100000 records

Since **cosin/io**-formatted 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 of 100000 records is allowed.

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
```

and the following group of lines:

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

are equivalent.

`var = value`
or
`var 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, for example, 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 also be an arbitrary number of spaces around the equality sign. The equality sign itself is optional. If the equality sign is not used, there must be at least one space separating the name and the value of the variable. For each scalar variable, a new record is required.

`var = text`
or
`var = 't1 t2'`

Instead of numerical values, strings may also be assigned to variables. They are specified either with, or without, quotation marks. The use of an equality sign is again optional. When using spaces, quotation marks, commas, semicolons, or exclamation marks within a string, the string has to be enclosed by quotation marks.

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 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 of 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/or lines. The dimensions of the vectors and matrices are only limited by the memory reserved by 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 separator. 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.

var = 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.

var = 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 the arithmetic expressions is described in the next section. If the expression comprises of spaces, quotation marks, commas, semicolons or exclamation marks, it has to be enclosed by quotation marks.

var = ?

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, when **cosin/io** is reading the file.

up to 500 data blocks

An input file may be sub-divided into 500 **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 $-signs at the beginning of a new data block, with a level not greater than the level of the preceding block, marks the beginning of a new block and at the same time marks the end of the preceding block. Different data blocks may comprise of 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 sub-blocks, 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 300 parameters within arithmetic expressions

In arithmetic expressions, up to 300 **parameters** (= variables, see below) may be used. The current values of the parameters are defined within one of the following data blocks: $parameters or $more_parameters. In doing so, a parameter may be replaced by an arithmetic expression, which itself may contain parameters. Clearly, the latter must be already defined by the time of evaluation of the first. See the `$parameters` example below.

`$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 actual values, after evaluation of all 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 by 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.

Document Revision: 2021-2-r24053

2 Arithmetic Expressions

operators
operations

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

- numerical 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, as used in most higher programming languages like C/C++, Fortran, or Matlab. Additional rules, which are not commonly used, are discussed below.
In **cosin/io**-formatted 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 below).

constants

In **cosin/io**-formatted input files, the rules applied to numerical constants are identical to those applied to variable values. A numerical 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.` , `30.0` , `+30.` , `3.0e1` , `300E-1`, etc.. Within a numerical constant, no blank spaces are allowed.

variables

In arithmetic expressions, **variables** (parameters) may be used instead of numerical constants. The variables and their respective values are stored in the **cosin/io**-input file within 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**(cosin sources & sinks), all **input signals** are automatically provided as variables. In all **cosin** applications, that generate **cosin/ip** outputs, the **plot signals** are also provided as variables.

`\string`

If a string, preceded by a backslash (\), 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 contain part of the string.

abc_123 =
ABC_123

The lengths of variable names are limited to 31 characters. Like in the C or Matlab programming language, the only permitted characters are letters, numerals and the underscore (_) character. It is not recommended to use a digit as the first character of a variable name. 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 = π/180
- r2d = 180/π

are predefined and may be used in every arithmetic expression. If used without the %-sign, these constants are subject to redefinition: the user may assign different values in the data blocks $parameters and $more_parameters. By prepending a '%', redefinition is avoided.

rnd
rndn
%rnd
%rndn

rnd and rndn are predefined variables, which generate different types of **random numbers** when used in an expression:

- rnd generates random numbers, **equally distributed** numbers in the interval [0,1],
- rndn generates **normally distributed** numbers, with a mean value of 0 and a standard deviation of 1.

Again, rnd and rndn can be redefined, in contrast to their counterparts with a leading '%', see above.

?

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

"

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 is 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 `!` 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
 same as
a < (b+c)
```

Arithmetic operations have priority over logical operations.

```
=
<
>
<=
>=
<>
eps=
```

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 of 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 & c
 same as
(a<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(.,.)
sign(.)

The following **mathematical and special 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)
- sign(x)

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.

`havsin(...)`
`scos(...)`
`ssin(...)`
`dwellsin(...)`

In addition to the standard mathematical functions, the following special functions are provided for convenience:

- `havsin(x,xmin,xmax,ymin,ymax)` is a smooth step function, based on the **haversine** function. It provides a differentiable transition of the output between y_{min} and y_{max}, for $x\epsilon\,[x_{min}, x_{max}]$
- `scos(x,xmin,dx,ymax)` is a single cosine cycle for $x\epsilon\,[x_{min}, x_{min} + dx]$ with amplitude y_{max}
- `ssin(x,xmin,dx,ymax)` is a single sine cycle for $x\epsilon\,[x_{min}, x_{min} + dx]$ with amplitude y_{max}
- `dwellsin(t,tstart,amp)` is the 'sine with dwell' function according to vehicle dynamics standard ECE 13-H, page 81, with amplitude amp (should be 1.5 * steering wheel angle at stationary 0.3g lateral acceleration).

int(.)
der(.)
pt1(.,.)
pd(...)
pid(...)
fishhook(...))

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].
- `pd(ya,yn,Kp,Td,umin,umax)` is a PD feed-back controller, if t appears as variable in the variable list. Parameters of `pd(...)`:
  - `ya` is the actual plant output,
  - `yn` is the nominal plant output,
  - `Kp` is the proportional gain,
  - `Td = Kd/Kp` defines the differential gain,
  - `umin` and `umax` are the controller output lower and upper bounds. `pd(...)` is an implementation of the general PD controller law:

    $$\overline{u} = K_p \left((y_n - y_a) + T_d \frac{d}{dt} (y_n - y_a) \right)$$

    $$u = max \left(u_{min}, min \left(u_{max}, \overline{u} \right) \right)$$
- `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:

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

    $$u = max \left(u_{min}, min \left(u_{max}, \overline{u} \right) \right)$$
- `fishhook(t,tstart,rollvel,amp)` is an implementation of the NHTSA fishhook steering maneuver:
  - `t` is the simulation time,
  - `tstart` is the start time of the maneuver,
  - `rollvel` is the current vehicle rolling velocity in deg/s,
  - `amp` is the steering amplitude (should be 6.6 * steering wheel angle at stationary 0.3g lateral acceleration).

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

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

- `pconst(file,datablock,x)` evaluates a piecewise constant function of a single independent variable x (cf. chapter 3). The interpolated data points are taken from data block 'datablock' of file 'file' . If 'file' equals '*', the current 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 3). The interpolated data points are taken from data block 'datablock' of file 'file'. If 'file' equals '*', the current 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 3) with linear extrapolation if argument is outside range of data points. The interpolated data points are taken from data block 'datablock' of file 'file' . If 'file' equals '*', then the current input file is used, where the function definition appears,
- `splic(file,datablock,x)` evaluates a cubic spline function of one independent variable x (cf. chapter 3) with constant end values if argument is outside range of data points. The interpolated data points are taken from data block 'datablock' of file 'file' . If 'file' equals '*', then the current input file is used, where the function definition appears,
- `poly(file,datablock,x)` evaluates a polynomial of one independent variable x (cf. chapter 3). The interpolated data points are taken from data block 'datablock' of file 'file'. If 'file' equals '*', the current 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 3). The interpolated data points are taken from data block 'datablock' of file 'file'. If 'file' equals '*', the current 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 3). The interpolated data points are taken from data block 'datablock' of file 'file'. If 'file' equals '*', the current 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 aa **'@'** 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 a '*' 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 to analyze a swept sine wave.

- `sweep(fstart,ffinal,Tf,t)` computes a sine wave with varying frequency and an amplitude of 1. The 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,
  - `Tf` is the duration of frequency variation,
  - `t` is the actual time (or independent argument)
- `swlin(fstart,ffinal,Tf,t)` is parameter compatible to `sweep`, but will vary the frequency at a constant rate ('linear sweep'),
- `frequ(fstart,ffinal,Tf,t)` is parameter compatible to `sweep`, and computes the actual frequency of the logarithmic sweep,
- `frlin(fstart,ffinal,Tf,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 (enclosed in single quotes, only necessary if it contains blank spaces). If a period ('.') as place-holder for the file is used, the file-name is determined by using the file-identifier `iscrf`,
  - `signalx` is the name of the plot-signal, as defined in the data file, which determines the values of the independent variable. The signal name may not contain any 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. A piecewise linear interpolation will be used.
- `meas(signal,xrel)` will read the signals in the file defined by file-identifier imeas. The signal of the independent variable is taken to be the relative position in the file (0.0 for the first signal value in the file, 1.0 for the last signal value, etc.). `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 may not contain any blank spaces. To generate a blank space in the signal name, use a period ('.') at the respective position in the string,
  - `xrel` is the relative position in the file, ranging from 0.0 to 1.0.

`inival(...)`
`dur(.)`
`mean(.,.)`
`rms(.,.)`
`mins(.,.)`
`maxs(.,.)`
`firsts(.,.)`
`lasts(.,.)`
`trig(...)`
`event(...)`

These functions extract certain signal properties for the input signals. In all functions, '`file`' is the name of the data file (enclosed in single quotes if it contains blank spaces) which holds the input signal/s. `signal` (or `signal_x`, `signal_y`) is the name of a plot-signal as defined in the data file from which information is to be extracted. The signal names may not contain any blank spaces. To replace a blank space, use a period ('.') at the respective position in the string. If the signal name only consists of a single period, the first signal in the data file will be used.

- `inival(file,signal_x,signal_y,tolerance,xsearch)` determines the initial mean value of `signal_y`. This value is determined by averaging the signal over an interval of `signal_x` that is defined by the last two parameters. `tolerance` specifies an absolute deviation of the y-signal from the first value, and and is used to determine the end of the averaging interval. `xsearch` sets a value of the x-signal, at which the averaging is to be started,
- `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`,
- `firsts(file,signal)` determines the first value of `signal`,
- `lasts(file,signal)` determines the last value of `signal`,
- `trig(file,signal_x,signal_y,tolerance,xsearch)` or `event(file,signal_x,signal_y,tolerance,xsearch)` determines the first value of `signal_x` at which the value of `signal_y` changes significantly. The meaning of the last two parameters is the same as in the function `inival(...)`.

`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**- or TeimOrbit-compatible data file (enclosed in single quotes, only necessary if it contains blank spaces),
  - `db` is the name of the data block the variable is to be searched in, and
  - `name` is the name of the data item, the value of which is to be returned.

Attention: if, in a parameter defining expression, `param` refers to the same parameter which is defined, an infinite recursion might happen, finally leading to a crash of the calling program.

`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, that has 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 returning values can 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 may not contain any 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 the slider is in its left-most position,
  - `sinit` is the initial slider value,
  - `smax` is the maximum slider value, which will be returned if the slider position is in it right-most position.
- `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 may not contain any 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 the slider position is in its left-most position,
  - `sinit` is the initial slider value,
  - `smax` is the maximum slider value, the rounded value of which will be returned if the slider position is in its right-most position.

3 Look-up Tables, Splines, and Polynomials

function types

cosin/io includes 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 a continuity of order zero is given at the respective data point. In addition to the function value, the simulation program may also request the evaluation of the first and second derivatives.
If either of the above function types is used by a simulation program, the corresponding data is 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. Function data is listed within the 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 for **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 be 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. For 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 an **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
1 2
2 5
^ 3-2*eps 7
4 2
5 3
^ 6+eps 7
7 5
```

has two kinks, one at $x = 3 - 2\epsilon$, and another one at $x = 6 + \epsilon$.

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. The 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 $y = p(x) = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n$, the corresponding polynomial data are defined by the vector of coefficients $a_0, a_1, a_2, \ldots, a_n$. Some or all components of this vector may be defined by arithmetic expressions, and all syntax rules for vector-valued data apply (as described above). The degree n of the polynomial is automatically recognized by counting 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, $i = 1, \ldots, n$
- a vector of values for the second independent variable y_k, $k = 1, \ldots, m$
- an array of values of the dependent variable z_{ik}, $i = 1, \ldots, n$, $k = 1, \ldots, m$.

These values are arranged as follows:

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

That is, the table looks like:

	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 **cosin/io** syntax rules apply for entering such a table in a data block, with the following amendment: the first row has one element less than all following rows.

Index