The Multi-Run Simulation Environment

SimEnv

User Guide for Version 2.11 (12-Jan-2010)

by M. Flechsig, U. Böhm, T. Nocke & C. Rachimow

Disclaimer of Warranty

The authors make no warranties, expressed or implied, that the programs and data contained in the software package and the formulas given in this document are free of error, or are consistent with any particular standard of merchantability, or that they will meet the requirements for any particular application. They should not be relied for solving a problem whose incorrect solution could result in injury to a person or loss of property. Applying the programs or data or formulas in such a manner is on the user's own risk. The authors disclaim all liability for direct or consequential damages from the use of the programs and data.

The Multi-Run Simulation Environment

SimEnv

User Guide for Version 2.11 (12-Jan-2010)

by

Michael Flechsig	flechsig@pik-potsdam.de
Uwe Böhm	boehm@pik-potsdam.de
Thomas Nocke	nocke@pik-potsdam.de
Claus Rachimow	rachimow@pik-potsdam.de

SimEnv on the Internet:
http://www.pik-potsdam.de/software/simenv/

Potsdam Institute for Climate Impact Research
Telegrafenberg
14473 Potsdam, Germany
Phone ++49 – 331 – 288 2604
Fax ++49 – 331 – 288 2640
WWW http://www.pik-potsdam.de

Contents

Tables

Figures

Examples

That is what we meant by science. That both question and answer are tied up with uncertainty, and that they are painful. But that there is no way around them. And that you hide nothing; instead, everything is brougth out into the open.

Peter Høeg, Borderliners
McClelland-Bantam, Toronto, 1995, p. 19

Executive Summary

SimEnv is a multi-run simulation environment that focuses on evaluation and usage of models with large and multi-dimensional output mainly for quality assurance matters and scenario analyses using sampling techniques.
Interfacing models to the simulation environment is supported for a number of model programming languages by minimal source code modifications and in general at the shell script level. Pre-defined experiment types are the backbone of SimEnv, applying standardised numerical sampling schemes for model parameters, initial or boundary values, or driving forces spaces. The resulting multi-run experiment can be performed sequentially or in parallel. Interactive experiment post-processing makes use of built-in operators, optionally supplemented by user-defined and composed operators. Operator chains are applied on model output and reference data to navigate and post-process in the combined sample and model output space. Resulting post-processor output data can be evaluated within SimEnv by advanced visualization techniques.

Simulation is one of the cornerstones in scientific research. The aim of the SimEnv project is to develop a toolbox oriented simulation environment that allows the modeller to handle model related quality assurance matters (Saltelli *et al.*, 2000 & 2004) and scenario analyses. Both research foci require complex simulation experiments for model inspection, validation and control design without changing the model in general.

SimEnv (Flechsig et.al, 2005) aims at model evaluation by performing simulation runs with a model in a co-ordinated manner and running the model several times. Co-ordination is achieved by pre-defined experiment types representing multi-run simulations.
According to the strategy of a selected experiment type for a set of so-called factors ***x*** which represent parameters, initial or boundary values, or drivers of a model ***M*** a numerical sample is generated before simulation. This sample corresponds to a multi-run experiment with the model. During the experiment for each single simulation run the factors *x* are adjusted numerically according to the sample and the factors' default values. Each experiment results in a sequence of model outputs for selected state variables ***z*** of the model *M* in the space of all addressed factors **{X}**. Model outputs can be processed and evaluated after simulation generally on the state space and experiment-type specifically on the factor space.

The following experiment types form the base of the SimEnv multi-run facility:

- Global sensitivity analysis
  Qualitative ranking of a large number of factors x with respect to their sensitivity on model output at random trajectories in the factor space {X}.
  For determination of the most important factors.
- Behavioural analysis
  Inspection of the model's behaviour in the factor space {X} by a discrete numerical sampling with a flexible inspection strategy for sub-spaces.
  For model verification, numerical validation, deterministic error analysis, deterministic control design, scenario analysis and spatial patch model applications.
- Local sensitivity analysis
  Determination of model (state variable's *z*) local sensitivity to factors *x*. Is performed by finite difference derivative approximations from *M*.
  For numerical validation purposes, model analysis, sub-model sensitivity.
- Monte Carlo analysis
  Factor space {X} sampling by perturbations according to probability density functions. Determination of moments, confidence intervals and heuristic probability density functions for state variables in the course of experiment post-processing.
  For error analysis, uncertainty analysis, verification and validation of deterministic models.
- Optimization
  Determination of optimal factor values by a simulated annealing method for a cost function derived from *z*.
  For model validation (system - model comparison), control design, decision making.

SimEnv makes use of modern IT concepts. Model preparation for interfacing it to SimEnv is based on minimal source code manipulations by implementing interface function calls into Fortran, C/C++, Python, Java, Matlab, Mathematica and GAMS model source code for the addressed factors and model output. Additionally, interfaces are available at shell script level and for supporting ASCII files.

In experiment preparation an experiment type is selected and equipped numerically by sampling the factor space. Experiment performance supports local, remote, and parallel / distributed hardware architectures to distribute work load of the single runs of the experiment.

Experiment specific model output post-processing enables navigation in the complex factor - model output space and interactive filtering of model output and reference data by application of operator chains. SimEnv supplies built-in operators and enables specification of user-defined and composed operators.

Result evaluation is dominated by application of pre-formed visualization modules using the visualization framework SimEnvVis of SimEnv.
SimEnv model output as well as experiment post-processing offer data interfaces for NetCDF, IEEE compliant binary and ASCII format for a more detailed post-processing outside SimEnv.

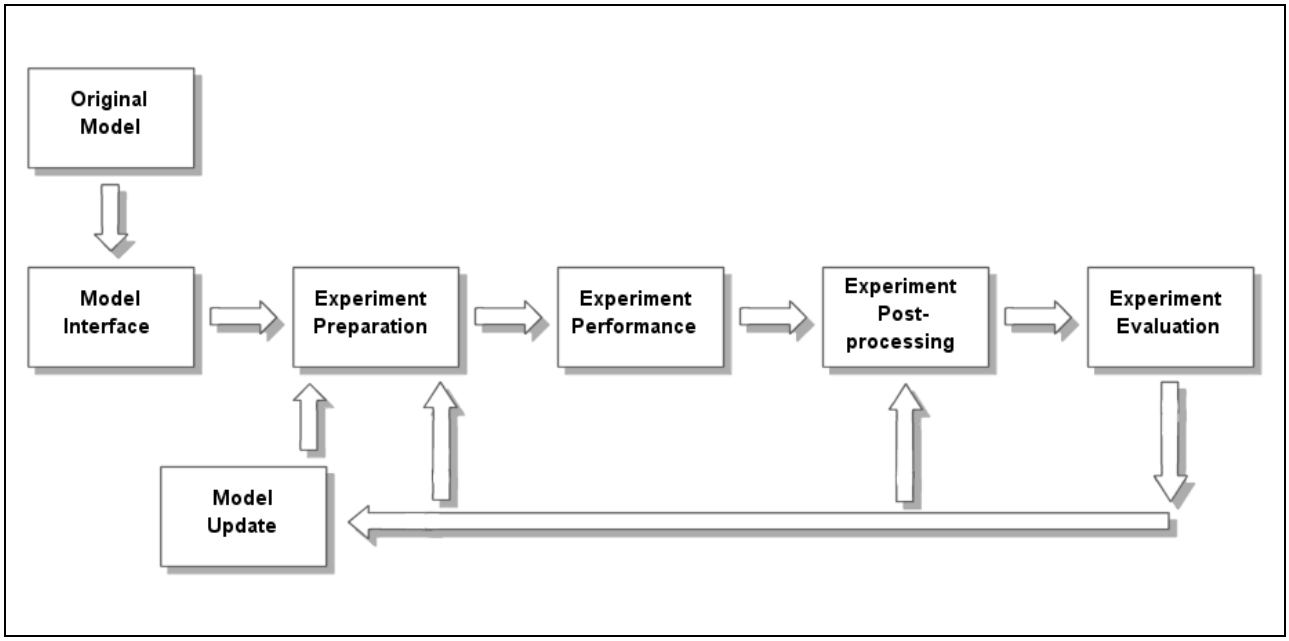


Fig. 0.1 *SimEnv system design*

SimEnv key features:

- Available for Unix and Linux platforms
- Support of key working techniques in experimenting with models:
  SimEnv enables model evaluation, uncertainty and scenario analyses in a structured, methodologically sound and pre-formed manner applying sampling techniques.
- Run ensembles instead of single model runs:
  Model evaluation by multi-run simulation experiments
- Availability of pre-defined multi-run simulation experiment types:
  To perform an experiment only the factors (parameters, initial values, drivers, ...) to experiment with and a strategy how to sample the factor space have to be specified.
- Simple model interface to the simulation environment:
  Model interface functions allow mainly to adjust an experiment factor numerically and to output model results for later experiment post-processing. Model interfacing and finally communication between the

model and SimEnv can be done at the model language level by incorporating interface function calls into model source code (C/C++, Fortran, Python, Java and Matlab: “include per experiment factor and per model output variable one additional SimEnv function call into the source code”) or can be done at the shell script level. Additionally, there are special interfaces for Mathematica and GAMS models.

- Support of distributed models:
  Independently on the kind distributed model components are interfaced to SimEnv and among each other the total model can be run within SimEnv.
- Parallelization of the experiment:
  This is a prerequisite for a lot of simulation tasks.
- Operator-based experiment post-processing:
  Chains of built-in, user-defined and composed operators enable interactive experiment post-processing based on experiment model output and reference data including general purpose and experiment specific operators. There is a simple interface to write user-defined and to derive composed operators.
- Graphical experiment evaluation:
  For post-processed model output
- Support of standard data formats:
  Output from the model as well from the post-processor can be stored in NetCDF or IEEE compliant binary format.

1 About this Document

In this chapter document conventions are explained. Within the whole document one generic reference example model is used to explain application of SimEnv. Examples are always located in grey boxes.

1.1 Document Conventions

Tab. 1.1 *Document conventions*

Character / string	Meaning
< ... >	angle brackets enclose a placeholder for a string
{ ... }	braces enclose an optional element
[... | ... | ...]	square brackets enclose a list of choices, separated by a vertical bar
' ... '	single quotation marks enclose a keyword or sub-keyword from user-defined files
" ... "	double quotation marks enclose the string-value of a sub-keyword from user-defined files
<nil>	stands for the empty string (nothing)
`monospace`	indicates SimEnv example code
blue_underlined	hyperlink in the document

Tab. 1.2 *Main placeholders in this document*

Placeholder	Description
<directory>	path to a directory
<factor_adj_val>	resulting adjusted value of a factor by <factor_smp_val> and <factor_def_val>
<factor_def_val>	default value of a factor as defined in <model>.edf
<factor_name>	name of a factor to experiment with as defined in <model>.edf
<factor_smp_val>	default value of a factor as defined in <model>.edf
<file_name>	name of a ASCII data file
<int_val>	integer value (e.g., -1234)
<model>	model name to start a SimEnv service with
<real_val>	real (float) value in integer (e.g., -1234), fixed point (e.g., -1234.) or floating point (scientific) (e.g., -0.1234e+4) notation
<simenv_res_char>	2-character experiment post-processor output file number 01, 02, ..., 99
<simenv_res_int>	integer post-processor output file number 1, 2, ..., 99
<simenv_run_char>	6-character single run number 000000, 000001, ... of an experiment
<simenv_run_int>	integer single run number 0, 1, ... of an experiment
<sep>	sequence of white spaces as item separators in user-defined and related files
<string>	any string
<val_list>	list of values in explicit or implicit notation according to Tab. 11.6
For post-processor operator descriptions only	
arg	general numerical argument (operand)
char_arg	character argument (operand), enclosed in single quotation marks
int_arg	integer constant argument (operand) ≥ 0
real_arg	real (float) constant argument (operand)

1.2 Example Layout

All examples in this document but for GAMS refer to a hypothetical global simulation **model world**. It describes dynamics of atmosphere and biosphere at the global scale over 200 years. Lateral (latitudinal and longitudinal) model resolution differs for different model implementations, temporal resolution is at decadal time steps. Additionally, atmosphere is structured vertically into levels. For more information on this generic model check Section 15.2.1.

The model world is assumed to map lateral and vertical (level) fluxes and demands that's why for computing state variables for all grid cells. However, in the model gridcell_f state variables are calculated for each grid cell without consideration of lateral fluxes.

Model implementation in a programming language <lng> results in a model world_<lng>.

Model state variable	Description	Defined on	Data type
atmo	aggregated atmospheric state	lat x lon x level x time	float
bios	aggregated biospheric state at land masses (defined between 84°N and 60°S latitude at land masses, i. e., without Antarctic)	lat x lon x time	float
atmo_g (not for model gridcell_f)	aggregated global state derived from atmo for level 1	time	int
bios_g (not for model gridcell_f)	aggregated global state derived from bios	-	int

Dynamics of all model variables depend on model parameters p1, p2, p3 and p4.

With this SimEnv release the following model implementations are distributed:

Model "auto" in name = semi-automated model interface	Model interface example for language <lng>	Resolution		
		lateral: lat x lon [deg x deg]	vertical: number of levels	temporal: number of time steps
world_f	Fortran	4 x 4	4: 1, 7, 11, 16	20
world_c	C	4 x 4	4: 1, 7, 11, 16	20
world_cpp	C++	4 x 4	4: 1, 7, 11, 16	20
world_py	Python	4 x 4	4: 1, 7, 11, 16	20
world_ja	Java	4 x 4	4: 1, 7, 11, 16	20
world_m	Matlab	4 x 4	4: 1, 7, 11, 16	20
world_sh	Shell script level	4 x 4	4: 1, 7, 11, 16	20
world_as	ASCII	4 x 4	4: 1, 7, 11, 16	20
world_f_auto	Fortran	4 x 4	4: 1, 7, 11, 16	20
world_sh_auto	Shell script level	4 x 4	4: 1, 7, 11, 16	20
world_f_1x1	Fortran	1 x 1	16: 1 - 16	20
world_f_05x05	Fortran	0.5 x 0.5	16: 1 - 16	20
gridcell_f	Fortran	without, implicitly by experiment as 4 x 4	4: 1, 7, 11, 16	20

Examples in this document are generally placed in grey-shaded boxes. Examples that are available from the example directory $SE_HOME/exa of SimEnv are marked as such in the lower right corner of an example box. To copy files from this directory use the SimEnv service simenv.cpy (cf. Tab. 10.4).

Example 1.1 *General example layout in the User Guide*
For Mathematica and GAMS models see Sections 5.6 and 5.7.

2 Getting Started

In this chapter a quick start tour is described. Without going into details the user can get an impression how to apply SimEnv and which files are essential to use the simulation environment.

- SimEnv is implemented under AIX-Unix at IBM's RS6000 and compatibles and SUSE-Linux at Intel-based platforms and compatibles. For detailed system requirements check Tab. 15.2 on page 161.
- Set the SimEnv home directory $SE_HOME and expand the PATH environment variable in the file $HOME/.profile by $SE_HOME/bin

  ```
  export SE_HOME=<se_home_path>
  export PATH=$SE_HOME/bin:$PATH
  ```

  <se_home_path> is the directory SimEnv is available from. For SE_HOME at PIK check Tab. 3.3 on page 9,,for the complete environment check Tab. 10.15 on page 139. Then apply the above setting by

  ```
  . $HOME/.profile
  ```
- Change to a directory with full access permissions. This is the SimEnv current workspace.
- Start

  ```
  simenv.hlp
  ```

  to acquire basic information on how to use SimEnv.
- Select a model implementation language <lng> to check SimEnv with the model world_<lng> from Example 1.1 on page 6:

<lng> =	f	for Fortran
	c	for C
	cpp	for C++
	py	for Python
	ja	for Java
	m	for Matlab
	sh	for shell script level
	as	for ASCII file

  For Mathematica models check Section 5.6 on page 36, for a GAMS model example check Section 5.7 on page 38.
- Start

  ```
  simenv.cpy      world_<lng>
  ```

  to copy the model world_<lng> model and experiment related files to the current workspace.
- Copy the file world.edf_3c to world_<lng>.edf
- Check

		for
• The SimEnv configuration file	**world_<lng>.cfg**	general SimEnv configurations
• The model output description file	**world_<lng>.mdf**	available model output variables
• The model	**world_<lng>.<lng>**	implementation of the model
• The model wrap shell script	**world_<lng>.run**	wrapping the model executable
• The experiment description file	**world_<lng>.edf**	experiment definition
• The post-processing input file	**world.post_3c**	post-processor result sequence

  Either
  - Start

    ```
    simenv.cpl      world_<lng>   -1   world.post_3c
    ```

    to run a complete SimEnv session:
    - Model and experiment related files will be checked
    - The experiment will be prepared
    - The experiment will be performed (select the login machine on request)

- Model output post-processing will be started for this experiment
  - With the post-processing input file world_post_3c and following
  - Interactively: Enter any result and finish post-processing by entering a single <return>
- Visualization of post-processed results will be started **(*)**
- Model or result output files will be dumped

or

- Start

simenv.chk	world_<lng>

to check model and experiment relate files.

- Start

simenv.run	world_<lng>

to prepare and perform a simulation experiment (select the login machine on request).

- Start

simenv.res	world_<lng> { new { <simenv_run_int> } }

to post-process the last simulation experiment for the whole run ensemble or for run number <simenv_run_int> and to create a new result file world_<lng>.res<simenv_res_char>.[nc | ieee | ascii] with the highest two-digit number <simenv_res_char>. <simenv_res_char> can range from 01 to 99.

- Start **(*)**

simenv.vis	world_<lng> { [latest | <simenv_res_char>] }

to visualize output from the latest post-processing session world_<lng>.res<simenv_res_char>.nc or that with number <simenv_res_char> with the highest two-digit number <simenv_res_char>.

- Start

simenv.dmp	world_<lng>	mod	|	more
simenv.dmp	world_<lng>	res	|	more

to dump a SimEnv model or post-processor output file.

- Check in the current workspace the

model interface	log-file	world_<lng>.mlog
native model terminal output	log-file	world_<lng>.nlog
experiment performance	log-file	world_<lng>.elog.

- Start

simenv.cln	world_<lng>

to wrap up a simulation experiment.

- Get the usage of any SimEnv service by entering the service command without arguments.
- To run other simulation experiments and/or output in other data formats modify
  - world_<lng>.cfg
  - world_<lng>.edf
  - world_<lng>.mdf
  - world_<lng>.run and/or
  - world_<lng>.<lng>
- To experiment with other models replace world_<lng> by <model> as a placeholder for the name of any other model.

(*): The visualization framework SimEnvVis of SimEnv does not belong to the standard SimEnv distribution. At PIK, for visualization set the DISPLAY environment variable accordingly. To get access permission for the PIK visualization server check in Section 10.2 on page 129 the SimEnv service

simenv.key	<user_name>

3 Version 2.11

This chapter summarizes differences between the current and the previous SimEnv release, limitations and bugs and their workarounds.

3.1 What is New?

Tab. 3.1 *SimEnv changes in Version 2.11*

Type	Check / see	On page	Description
update	Section 5.5	34	Regular interface for Matlab models including *simenv_slice*
new	Section 5.5	34	Interface function *simenv_slice* now available for Python and Java models
new	Section 10.1	125	New sub-keyword *slices* in <model>.cfg for applying interface function *simenv_slice* in user models
new	Section 10.1	125	New sub-keywords *include_runs* and *exclude_runs* in <model>.cfg to flexibly define the set of single runs to be performed during an experiment
update	Section 6.3	56	Behavioural analysis (<model>.edf): control checking of the sample defined in an external file (specific comb file <file_name>) by additional keys *strict* and *nonstrict*
update	Section 5.7.3	43	New logfile policy for GAMS models
update	Section 5.7.2	39	Sub-keyword *time_limit* in <model>.gdf for GAMS models again active
			Bug fixes

Tab. 3.2 *User actions to upgrade to Version 2.11*

Upgrade type	Upgrade action
mandatory	Re-link interfaced models and user-defined operators

Tab. 3.3 *SimEnv availability at PIK machines*

Machine	SE_HOME=
bs08	/usr/local/simenv
Iplex cluster	/iplex/01/sys/applications/simenv
viss	/usr/local/simenv

3.2 Limitations / Problems and Their Workarounds

Tab. 3.4 *Limitations / problems and their workarounds in Version 2.11*

Where Limitation / Problem Workaround	Description
Where	Overall
Limitation	Current SimEnv technical limitations as specified in Tab. 15.3 on page 162
Workaround	None
Where	Overall but visual result evaluation
Limitation	Without graphical user interface
Workaround	None
Where	Experiment performance: Model output to NetCDF
Problem	Check on undefined model output results in noticably additional CPU-time consumption. Example: Per single run, a check of 8 Mill of real*8 values takes additionally 80 sec for single nc-file model output and additionally 200 sec for common nc-file output.
Workaround	Specify in <model>.cfg for the sub-keyword ‘message_level’ the value = “error”
Where	Experiment performance: Experiment type optimization
Limitation	Cannot be performed under load leveler control in in sub-modes par and dis
Workaround	Perform optimization experiment in sequential mode
Where	Experiment performance: Experiment type optimization
Limitation	The initial seed for the optimization technique is fixed. That’s why the algorithm results for the same optimization problem always in the same sampled sequence in the factor space
Workaround	None

3.3 Known Bugs and Their Workarounds

Tab. 3.5 *Known bugs and their workarounds in Version 2.11*

Where Bug Workaround	Description
Where	Experiment performance: Distributed models (structure = 'distributed' in <model>.cfg) Model output to NetCDF
Bug	May not store all model output
Workaround	Specify IEEE model output in <model>.cfg
Where	Experiment performance: Under load leveler LoadL control in distributed sub-mode dis:
Bug	Experiment may not come to an end.
Workaround	Kill processes simenv_run_dismco and simenv_run_dismco_aux. Afterwards, **restart** the experiment by SimEnv service simenv.rst with an other mode / sub-mode.
Where	Experiment performance: Under load leveler LoadL control in distributed sub-mode dis: Monte Carlo experiment with stopping rule
Bug	Experiment does not come to an end.
Workaround	Run experiment with an other mode / sub-mode.

4 Experiment Types

SimEnv supplies a set of pre-defined multi-run experiment types. Each experiment type addresses a special experiment class for performing a simulation model several times in a co-ordinated manner. In this chapter an overview on the available experiment types is given from the viewpoint of system's theory.

4.1 General Approach, Computational Costs

SimEnv supplies a set of pre-defined multi-run experiment types, where each type addresses a special multi-run experiment class for performing a simulation model or any algorithm with an input - output transition behaviour.
In the following, the general SimEnv approach will be described for time dynamic simulation models, because this class forms the majority of SimEnv applications. All information can be transformed easily to any other algorithm.

Based on systems' theory, each time dynamic model *M* can be formulated - without limitation of generality - for the time dependent, time discrete, and state deterministic case as

$$M: \quad Z(t) = ST\ (\ Z(t-\Delta t)\ ,...,\ Z(t-k*\Delta t)\ ,\ P\ ,\ IX(t)\ ,\ Z_0\ ,\ B\ )$$

with

ST	state transition description
Z	state variables' vector
P	parameter vector
IX	input (driving forces) vector
Z_0	initial value vector
B	boundary value vector
t	time
Δt	time increment
k	time delay

The output vector *Y* is a function of the state vector *Z*, parameters *P*, drivers *IX*, and initial values Z_0:

$$Y(t) = OU\ (\ Z(t)\ ,\ P\ ,\ IX(t)\ ,\ Z_0\ ).$$

Model behaviour *Z* is determined for fixed *k* and Δt by state transition description *ST*, parameters *P*, driving forces *IX*, initial values Z_0, and boundary values *B*. Manipulating and exploring model behaviour in any sense means changing these four model components. While state transition description *ST* reflects mainly model structure and is quite complex to change, each component of the driving forces vector *IX* normally is a time-dependent vector.

Introduction of additional technical parameters / triggers P_{tech} can reduce the complexity of handling a model with respect to the five model components, described above: Changes in state transition description *ST* can be pre-determined in the model by assigning values of a technical / trigger parameter p_{tech} to apply for example alternative model structures, sub-structures, processes formulations, resolutions, which are triggered by these values.
Additionally, each component of the driving forces vector *IX* can be combined with technical parameters in different ways:

- By selecting special driving forces dependent on the technical value
- By manipulating the driving forces with the parameter value
  (e.g., as an additive or multiplicative increment)
- By parametrizing the shape of a driving force

When this has been done, the model behaviour finally depends only on the parameters *P*, the initial values Z_0, and the boundary values *B*. From the methodical point of view there is no difference between parame-

ters, initial values and boundary values, because all are considered as constant during one model run. That is why in SimEnv all the four model components parameters, drivers, initial values and boundary values are lumped together and the term **factor** stands as a placeholder for them. An often used synonym for "factor" is "input". All factors form the factor space X:

$$X = \{ P, IX , Z_0 , B \}$$

and

$$Z = ST(X).$$

In the following,

$$X_k = (x_1 ,..., x_k) \qquad k > 0$$

stands for a subset of the factor space X that spans up a k-dimensional sub-space of X by selected model factors (x_1 ,..., x_k) from X and

$$X_{k,n} = \begin{pmatrix} x_{11} & \dots & x_{1n} \\ \dots & & \dots \\ x_{k1} & \dots & x_{kn} \end{pmatrix} = (\hat{}X_1 ,..., \hat{}X_n) \qquad k > 0,\ n > 0$$

stands for a numerical sample for X_k of size n and finally for k*n values representing in any sense the sample space X_k.
In the set of all samples $X_{k,1}$ $X_{k,1}$ is the default (nominal) numerical factor constellation for the model *M* as normally defined in the model source code.
If { } $_n$ denotes the dynamics of the model *M* over a sample of size n then it holds:

$$\{ Z \}_n = \{ ST(\hat{}X_1) ,..., ST(\hat{}X_n) \}.$$

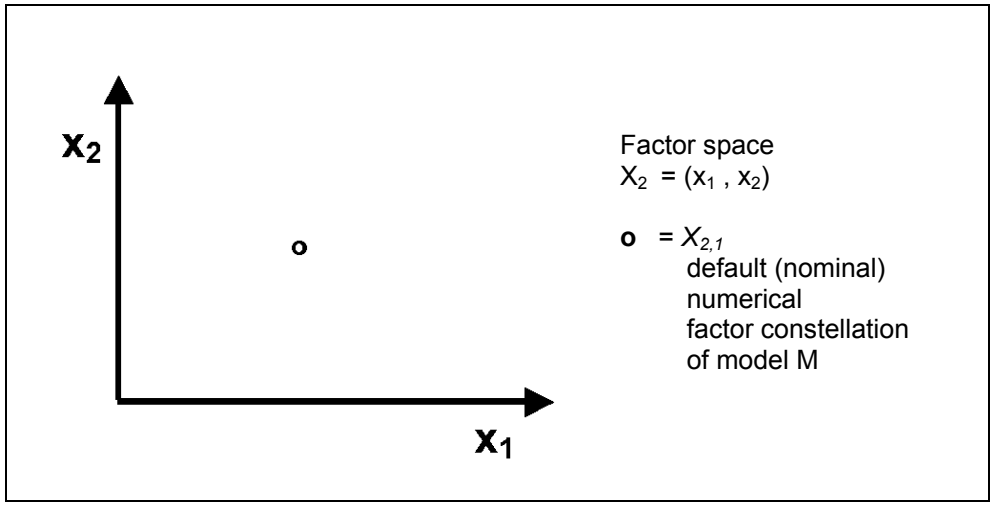


Fig. 4.1 *Factor space*

SimEnv supports different sampling strategies and the performance of multi-run experiments where k factors are adjusted numerically for each of n single simulation runs according the generated sample and the default (nominal) values of the factors. Central goal is to study the dependency of the model dynamics in the factor space. For simulation purposes in SimEnv experimentation with the model *M* over $X_{k,n}$ is based on the assumption that dynamics of *M* for each representative from the sample is independent from all other representatives, which is fulfilled in general. This results in the possibility to form a run ensemble for performing the model *M* with n single model runs from the sample $X_{k,n}$.

SimEnv experiment types differ in the way the sample space X_k is sampled to get $X_{k,n}$. There are deterministic and non-deterministic sampling strategies that offer a broad range of techniques for

- Experimentation with models
- Post-processing model output results

- Interpreting results with respect to uncertainty and sensitivity matters of models.

The experiment types are described in detail in the following Sections. They are ordered in a preferred sequence which should be used for a best results in assessing any model. Tab. 4.1 provides an overview on the experiment types together with their computational costs. The computational cost of a resulting experiment from an experiment type is the number of single model runs to perform.

Tab. 4.1 *Experiment types and their computational costs*
N denotes the number of Monte Carlo runs

Experiment Type	Description	Computational Costs (k factors)
global sensitivity analysis	Qualitative ranking of a large number of factors with respect to their sensitivity on model output at random trajectories in the factor space. For determination of the most important factors.	(5 … 10) * (k+1)+1
behavioural analysis	Inspection of the model's behaviour in the factor space by a discrete numerical sampling with a flexible inspection strategy for sub-spaces. For model verification, numerical validation, deterministic error analysis, deterministic control design, scenario analysis and spatial patch model applications.	experiment dependent
local sensitivity analysis	Determination of model (state variable's) local sensitivity to factors. Is performed by finite difference derivative approximations from the model. For numerical validation purposes, model analysis, sub-model sensitivity.	2*k+1
Monte Carlo analysis	Factor space sampling by perturbations according to probability density functions. Determination of moments, confidence intervals and heuristic probability density functions for state variables in the course of experiment post-processing. For error analysis, uncertainty analysis, verification and validation of deterministic models.	N+1
optimization	Determination of optimal factor values by a simulated annealing method for a cost function derived from state variables. For model validation (system - model comparison), control design, decision making.	unpredictable

4.2 Global Sensitivity Analysis

The guiding philosophy of a global sensitivity analysis is to determine these factors that influence a model state z the most and to distinguish them from these factors that are negligible. Contrary to a local sensitivity analysis, during a global sensitivity analysis the entire space where the factors may vary is considered.
The global sensitivity analysis in SimEnv applies the method of Morris (1991) in its modification by Campolongo *et al.* (2005). Its main approach is to derive qualitative global sensitivity measures for all factors by computing a statistics on a series of local sensitivity measures, the so-called elementary effects. The result of this analysis is a ranking of the factors in order of importance with respect to the model state z.

The modified Morris method is as follows (cf. also Fig. 4.2):

- Start for each factor with the determination of the so-called sensitivity range where the factor may vary. All k factors span up with their sensitivity ranges a k-dimensional cube.
- Sub-divide this cube into a regular k-dimensional p-level grid by determining within the sensitivity range of each factor p-2 equidistant grid points. Together with the bounds from the sensitivity range this results in p equidistant points for each factor.

- Select at the p-level grid randomly a starting grid point $x = (x_1, \ldots, x_k)$ and at the grid randomly a next-neighbour (adjacent) grid point $(x_1, \ldots, x_{i-1}, x_i+\Delta_i, x_{i+1}, \ldots, x_k)$ that differs from the starting grid point in exactly one factor x_i (i=1 ,…, k).
- Compute from these two grid point the elementary effect for the model state variable z
  $d_i(x,z) = z(x_1, \ldots, x_{i-1}, x_i+\Delta_i, x_{i+1}, \ldots, x_k) - z(x_1, \ldots, x_k)$
- Proceed by randomly selecting a new next-neighbour grid point to the old next-neighbour grid point for another elementary effect $d_j(x,z)$ $(i \neq j)$ until k+1 points (including the starting point) are sampled. Such a series of k+1 point is called a trajectory. For one trajectory k elementary effects $d_i(x,z)$ (i=1 ,…, k) can be determined by two consecutive points.
- Determine randomly r trajectories in this way finally resulting in r elementary effects $d_i(x,z)$ for each i in 1 ,…, k.
- Consider distributions

  $F_i^{abs} = \{ |d_i(x,z)| \}$ and compute mean $\mu_i^{abs} = \sum|d_i(x,z)| / r$ for i=1 ,…, k

  $F_i = \{ d_i(x,z) \}$ and compute variance $\sigma_i = \sum (d_i(x,z) - \sum d_i(x,z) / r)^2 / (r - 1)$ for i=1 ,…, k
- Consider in the (μ^{abs}, σ) plane the points (μ_i^{abs}, σ_i), (i=1 ,…, k):
  a high value of μ_i^{abs} with respect to the other μ_j^{abs}
  indicates an important overall influence of the factor x_i on the model state z
  a high value of σ_i with respect to the other σ_j
  indicates that the factor x_i is involved in interactions with other factors or
  indicates that the effect of x_i on the model state z is non-linear
  (Saltelli *et al.*, 2004)

Note that this approach differs from Morris and Campolongo et al. as follows:
- An elementary effect is derived from adjacent points of the p-level grid. The original method allows for elementary effects $d_i(x,z)$ from points differing in a pre-determined multiple of Δ_i.
- Originally, elementary effects are divided by Δ_i and/or its pre-determined multiple. To make the method work also for factors that influence model output with different orders of magnitude and for different orders of magnitude of the sensitivity ranges the global sensitivity algorithm in SimEnv does not divide.

According to Saltelli *et al*., (2004) as a rule of thumb, normally p ranges between 4 and 6 and r around 10.

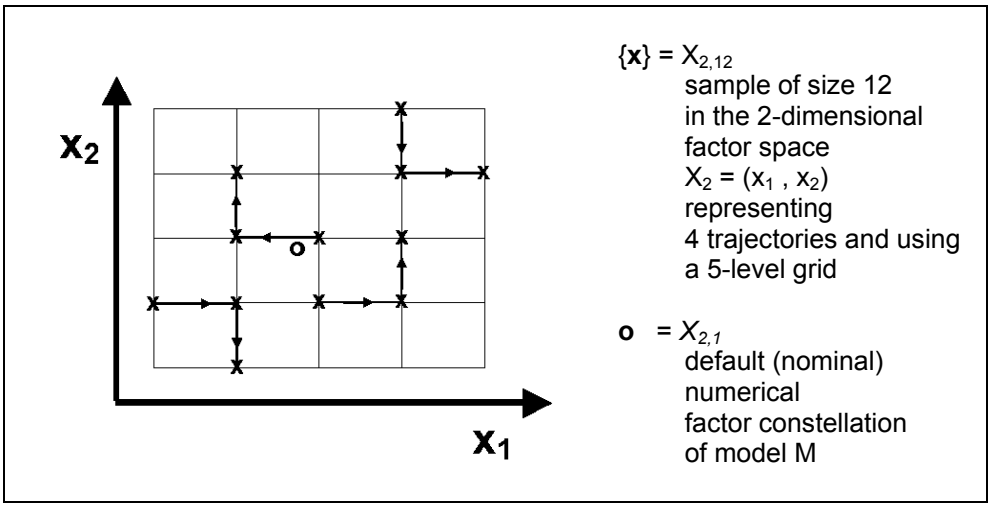


Fig. 4.2 *Sample for a global sensitivity analysis*
The arrows indicate the sequence how sampling points were generated for each trajectory.

4.3 Behavioural Analysis

Behavioural analysis uses a deterministic strategy to sample X_k. It is the inspection of the model in the factor space X_k where inspection points are set in a regular and well structured manner.

Behavioural analysis can be interpreted and used in different ways:

- For scenario analysis:
  to show how model behaviour changes with changes of factor values
- For numerical validation purposes:
  to determine factor values in such a way that the output vector matches with measurement results of the real system
- For deterministic error analysis:
  to analyse how the model error is dependent on factor errors
- For a simulation-based control design:
  to determine factor values in such a way that a goal function becomes an extreme

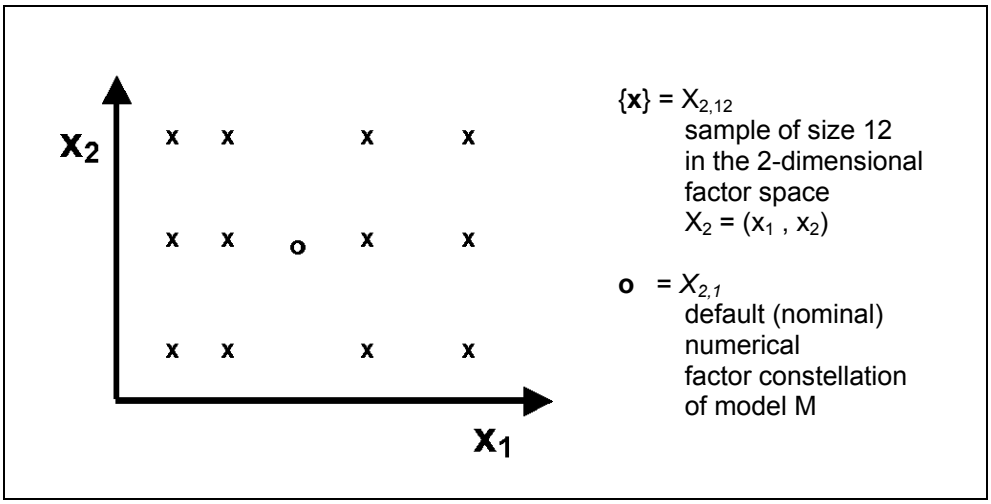


Fig. 4.3 *Sample for a behavioural analysis*

SimEnv behavioural analysis sampling strategy is a generalization of the one-dimensional case for X_1, where the model behaviour is scanned in dependence on deterministic sample of one factor x_1. The general case for X_k demands a strategy for scanning m-dimensional spaces in a flexible manner. Based on the predecessors of SimEnv (Wenzel *et al.*, 1990, Wenzel *et al.*, 1995, Flechsig, 1998) subspaces of the m-dimensional factor space can be scanned on the subspace diagonal (parallel in a one-dimensional hyperspace) or completely for all dimensions (combinatorially on a grid) and both techniques can be combined. Besides this regular scanning method an irregular technique is possible.

The resulting number of single simulation runs for the experiment depends on the number of factor samples per dimension of the scanned factor space and from the selected scanning method. An experiment is described by the names of the involved factors, their numerical sampling values and their combination (scanning method). Experiment post-processing can resolve the scanning method again and output results as projections on multi-dimensional factor subspaces.

Fig. 4.4 describes the regular scanning technique by an example. In the left scheme (a) the two-dimensional factor space $X_2 = (p_1 , p_2)$ is scanned combinatorially, resulting in 4*4 = 16 model runs, while the middle scheme (b) represents a parallel scanning of these two factors at the diagonal by 1+1+1+1 = 4 model runs. The scheme (c) at the right side shows a complex scanning strategy of the 3-dimensional factor space $X_3 = (p_1 , p_2 , p_3)$ with (1+1+1+1)*3 = 12 model runs. Each filled cross **x** in Fig. 4.4 represents a sample point in the factor space and finally a single model run of the experiment.

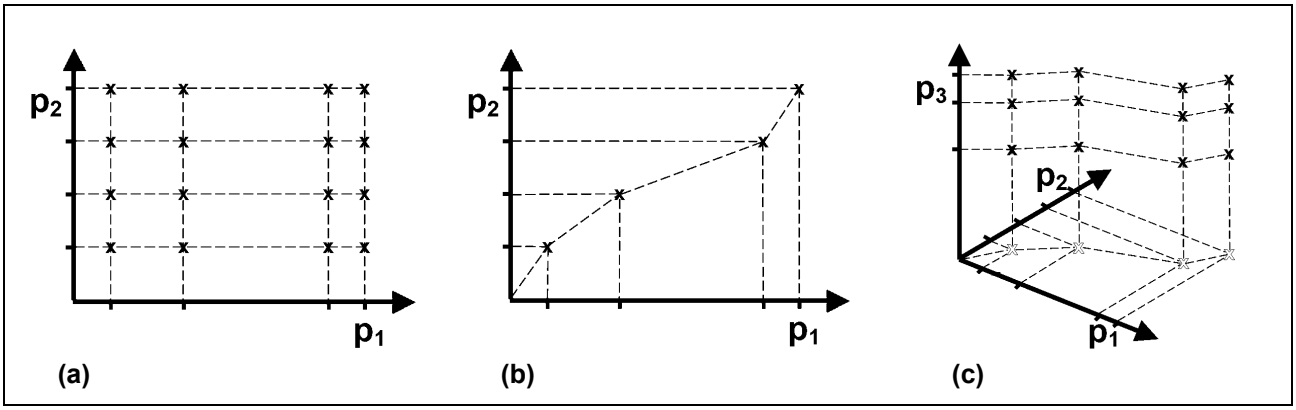


Fig. 4.4 *Behavioural analysis: Scanning multi-dimensional factor spaces*

4.4 Local Sensitivity Analysis

Local sensitivity analysis uses a deterministic sampling stategy in ε-neighbourhoods of the numerical default constellation $X_{k,1}$ of the model M. For each value x_i from the default (nominal) factor constallation $X_{k,1}$ and each ε_j from the ε-neighbourhoods (ε_1 ,…, ε_m) two members (x_1 ,..., x_{i-1} , $x_i \pm \varepsilon_j$, x_{i+1} ,..., x_k) of the resulting sample are generated. The sample size n is given by 2*m*k. Running the model for this sampling set serves to determine sensitivity functions.

In classical systems' theory, model sensitivity of a model state variable z with respect to a factor x is the partial derivative of z after x: $\delta z/\delta x$. In the numerical simulation of complex systems a finite sensitivity function is preferred, because it can be obtained without model enlargements or re-formulations. It is a linear approximations of the classical model sensitivity measure (Wierzbicki, 1984). Contrary to a global sensitivity analysis a local one covers the model's sensitivity in the neighbourhood of the default (nominal) factor constellation.

Local sensitivity measures as well as measures which reflect model output linearity and/or symmetry nearby $X_{k,1}$ can be used for localizing modification-relevant model parts as well as control-sensitive factors in control problems. On the other hand, identification of robust parts of a model or even complete robust models makes it possible to run a model under internal or external disturbances. Sensitivity analysis in SimEnv experiment post-processing is based on finite sensitivity, linearity, and symmetry measures, which are defined as in Tab. 4.2.

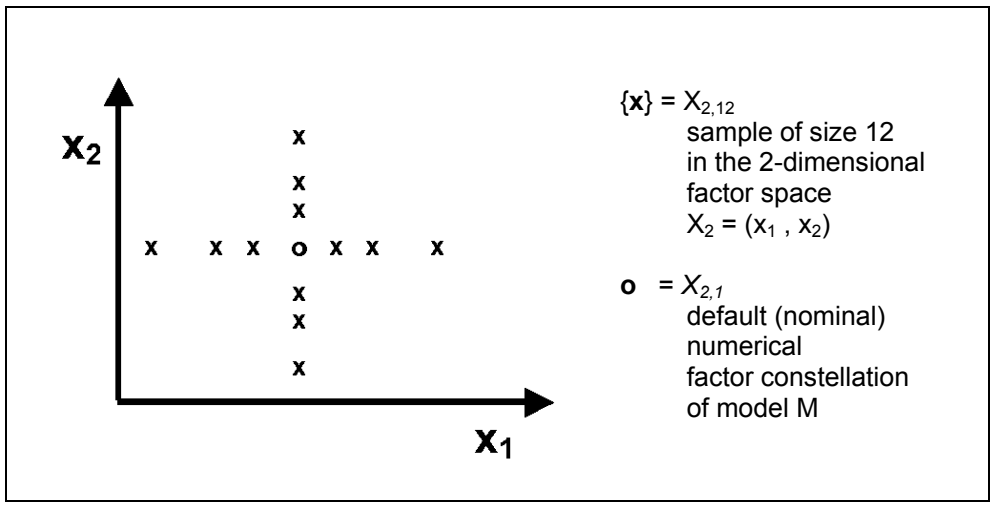


Fig. 4.5 *Sample for a local sensitivity analysis*

Tab. 4.2 *Local sensitivity, linearity, and symmetry measures for a state variable z, a selected factor x from $X_{k,1}$ and a selected value ε from (ε_1 ,…, ε_m)*

Local measure	Definition	
	Absolute measure	**Relative measure**
sensitivity measure	$sens\_abs(z,\pm\varepsilon) = \frac{z(t \pm \varepsilon) - z(t)}{\pm \varepsilon}$	$sens\_rel(z,\pm\varepsilon) = sens\_abs(z,\pm\varepsilon) \frac{t}{z(t)}$
linearity measure	$lin\_abs(z,\varepsilon) = \frac{(z(t+\varepsilon) - z(t)) + (z(t-\varepsilon) - z(t))}{\varepsilon}$	$lin\_rel(z,\varepsilon) = lin\_abs(z,\varepsilon) \frac{t}{z(t)}$
symmetry measure	$sym\_abs(z,\varepsilon) = \frac{z(t+\varepsilon) - z(t-\varepsilon)}{\varepsilon}$	$sym\_rel(z,\varepsilon) = sym\_abs(z,\varepsilon) \frac{t}{z(t)}$

Accordingly, local measures of the model with respect to a factor are always expressed as a measure of a model's state variable z, usually at a selected time step within a surrounding neighborhood ε of a factor value t. That is why the conclusions drawn from a local sensitivity analysis are only valid locally at $X_{k,1}$ with respect to the whole factor space X_k. Additionally, local measures only describe the influence of one factor x_i from the whole vector X_k on the model's dynamics.

As stated above, the sensitivity measures reflect the classical sensitivity functions in a neighborhood of $X_{k,1}$. The larger the absolute value of the measure the higher is the influence of an incremental change of the factor x on the model output z. The linearity measures map the linear behaviour of z nearby $X_{k,1}$. If the linear measure is zero z shows a linear behaviour with respect to x. The symmetry measures measures map the symmetric behaviour of the z nearby $X_{k,1}$. If the symmetry measure is zero z shows a symmetric behaviour with respect to x. The larger the absolute values of the latter two measures the higher is the nonlinear / non-symmetric behaviour of z with respect to x.

The absolute measures are best suited to compare the influence of different factors {x} on the same state variable z while due to their normalization factor the relative measures enable comparison of the influence of one factor x on different state variables {z}.

From the local measures of table Tab. 4.2 additional measures can be derived on demand, e.g., abs(sym_abs(z, ε)).

A local sensitivity experiment is described by the names of the factors x to be involved and the increments ε. The number of runs for the experiment results from the number of factors and increments: two runs per factor for each increment plus one run with the default values of the factors. Local sensitivity functions are calculated during experiment post-processing.

4.5 Monte Carlo Analysis

Monte Carlo analysis uses a non-deterministic strategy to sample $X_{k,n}$. A Monte Carlo experiment in SimEnv is a perturbation analysis with pre-single run factor perturbations.

Theoretically, with a Monte Carlo analysis moments of a state variable z can be computed as

$$M^{(m)}\{z\} = \int_{X_k}^{\cdots} z(X_k)^m * pdf(X_k)\, dX_k$$

with

$z(X_k)$	state variable z as a function of X_k
$pdf(X_k)$	probability density function of X_k
$M^{(m)}\{z\}$	m-th moment of the state variable z with respect to the probability density function pdf

By interpreting the probability density function pdf(X_k) as the error distribution in the factor space X_k it is possible to study error propagation in the model. On the other hand Monte Carlo analysis can be interpreted as a stochastic error analysis, if there are measurements of the real system for z.

For a numerical experiment in SimEnv it is assumed that the probability density function pdf(X_k) can be decomposed into independent probability density functions pdf_i for all factors x_i of X_k:

$$pdf(X_k) = \prod_{i=1}^{k} pdf_i(x_i)$$

and the k-dimensional integral is approximated by a sequence of n single simulation runs of the model where the numerical factor values x_{ij} of t_i ($1 \le i \le k$, $1 \le j \le n$) are sampled according to the probability density function pdf_i.
On the basis of these assumptions, the statistical measures in Tab. 4.3 can be computed during performance of an experiment post-processing session from a Monte Carlo analysis with n simulation runs resulting in n realizations z_1 ,..., z_n of the model's state variables z, z1 and z2:

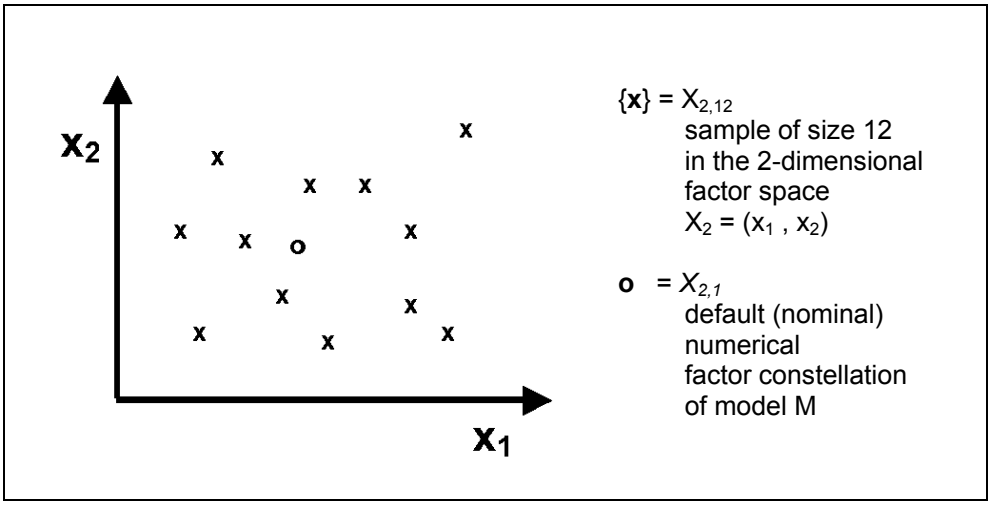


Fig. 4.6 *Sample for a Monte Carlo analysis*

Tab. 4.3 *Statistical measures*

(*)*: indices for sums* Σ*, products* Π *and extremes run from 1 to n:* $\sum_{i=1}^{n}$, $\prod_{i=1}^{n}$, $\min_{i=1,...,n}$, $\max_{i=1,...,n}$

Statistical measure	Definition (*)
minimum	min(z) = min (z_i)
maximum	max(z) = max (z_i)
sum	sum(z) = $\Sigma\, z_i$
arithmetic mean	avg(z) = $\Sigma\, z_i$ / n
variance	var(z) = $\Sigma\, (z_i - avg(z))^2$ / (n - 1)
skewness	skw(z) = $\Sigma\, (z_i - avg(z))^3$ / (n * $var(z)^{3/2}$)
kurtosis	krt(z) = $\Sigma\, (z_i - avg(z))^4$ / (n * $var(z)^2$) - 3
range	rng(z) = max(z) – min(z)
geometric mean	avgg(z) = $(\Pi\, z_i)^{1/n}$

Statistical measure	Definition (*)
harmonic mean	agvh(z) = n / $\Sigma(1 / z_i)$
weighted mean	avgw(z) = $\Sigma z_i * w_i / \Sigma w_i$ w : weight
correlation	cor(z1,z2) = $\Sigma (z1_i - avg(z1)) * (z2_i - avg(z2)) / \sqrt{\Sigma (z1_i - avg(z1))^2 * \Sigma (z2_i - avg(z2))^2}$
covariance	cov(z1,z2) = $\Sigma (z1_i - avg(z1)) * (z2_i - avg(z2)) / (n - 1)$
linear regression coefficient to forecast z2 from z1	reg(z1,z2) = $(\Sigma (z1_i - avg(z1)) * (z2_i - avg(z2))) / (\Sigma (z1_i - avg(z1))^2)$ It is: z2 = reg(z1,z2) * z1 + avg(z2) – reg(z1,z2) * avg(z1) + error
median	med(z) = middle value from increasingly ordered { z_i } (n = odd) mean of the two middle values from { z_i } (n = even)
quantile	$qnt^{(p)}(z)$ = that value from increasingly ordered { z_i } which corresponds to a cumulative frequency of n*p/100 $qnt^{(50)}(z)$ = med(z)
confidence interval boundaries	$cnf^{(\alpha)}(z) = avg(z) \pm t_{\alpha,n-1} \sqrt{var(z) / n}$ α : probability of error $t_{\alpha,n}$: significance boundaries of Student distribution
heuristic probability density function	$hgr^{(class)}(z)$ = number of z_i with $class_{min} \le z_i < class_{max}$ $class_{min}$, $class_{max}$: boundaries of equidistant classes

Tab. 4.4 summarizes these probability density functions that are pre-defined in SimEnv for factors to be perturbed. Additionally, SimEnv offers to import random number samples in the course of experiment preparation.

Tab. 4.4 *Probability density functions*

Distribution	Short-cut	Probability density function pdf	Distribution parameters
uniform	U(a,b)	$pdf(x) = \frac{1}{b-a}$ if $x \in [a,b]$ pdf(x) = 0 otherwise	a lower boundary b upper boundary > a it is: mean = (a+b) / 2 standard deviation = $\sqrt{(b-a)^2 / 12}$
normal	$N(\mu,\sigma^2)$	$pdf(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$	μ mean σ standard deviation > 0
lognormal	$L(\mu,\sigma^2)$	$pdf(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$ if x > 0 pdf(x) = 0 otherwise	μ σ > 0 it is: $\ln(x) \sim N(\mu,\sigma^2)$
exponential	$E(\mu)$	$pdf(x) = \frac{1}{\mu} \exp\left(-\frac{x}{\mu}\right)$ if x > 0 pdf(x) = 0 otherwise	μ mean > 0 it is: standard deviation = μ

The number of runs to be performed during a Monte Carlo analysis has to be specified. An experiment is described by the factors involved in the analysis, their distribution and the appropriate distribution parameters.
Optionally, a stopping rule is helpful to limit the number of simulation runs in an experiment. In a stopping rule statistical measures from model output z of all performed single runs are calculated during the experiment after each single model run to decide whether to stop the whole experiment. SimEnv supplies a simple rule-of-thumb stopping rule from Schuyler (1997), using the standard error of mean statistic

$$\sqrt{\text{var}(z) / n} \qquad \text{with n = number of already performed single runs}$$

and checks it against the mean avg(z).

4.6 Optimization

The optimization experiment in SimEnv uses a stochastic strategy to sample X_k. It is the only experiment type where the sample is generated during experiment performance and not at experiment preparation. The general approach of optimization is to find the global minimum of a cost function (synonym: objective function)

$$F(Z) = F(ST(X_k))$$

that depends on model's state variables Z and consequently on the experiment factors $X_k = (x_1, ..., x_k)$:

$$\begin{array}{ll} \text{minimize} & F(Z(x_1, ..., x_k)) \\ \text{subject to} & x_{i\,min} \le x_i \le x_{i\,max} \quad \text{for } i = 1, ..., k \end{array}$$

Often, F represents a distance measure in a specific metric between selected model state variables and reference data (measurement values of the real system or simulation results from an other model). Consequently, optimization can be used for model validation and control design to find optimal values of model factors in such a way that model state variables are close to reference data. In SimEnv the cost function is specified in experiment preparation as a single run result formed from model output (and reference data) where an operator chain is applied on (cf. Section 6.6 and Chapter 8). The value of the cost function is calculated directly after the current single run has been performed.

SimEnv uses a gradient free optimization approach that is called **"Simulated Annealing"** and is a generalization of a Monte Carlo method for examining the state equations of n-body systems. The concept is based on the manner in which metals recrystalize in the process of annealing. In an annealing process a melt, initially at high temperature Temp and disordered, is slowly cooled so that the system at any time is approximately in thermodynamic equilibrium. As cooling proceeds, the system becomes more ordered and approaches a "frozen" ground state at Temp = 0. Hence the process can be thought of as an adiabatic approach to the lowest energy state E. If the initial temperature of the system is too low or cooling is done insufficiently slowly the system may become quenched forming defects or freezing out in metastable states (i.e. trapped in a local minimum energy state).

The annealing scheme is that an initial state of a thermodynamic system is chosen at energy E and temperature Temp, holding Temp constant the initial configuration is perturbed and the change in energy dE is computed. If the change in energy is negative or zero the new configuration is accepted. If the change in energy is positive it is accepted with a probability given by

$$p = \exp(-dE/(k_B * Temp))$$

where k_B denotes the Boltzmann constant. This process is then repeated sufficient times to give good sampling statistics for the current temperature, and then the temperature is decremented and the entire process repeated until a frozen state is achieved at Temp = 0.

By analogy the generalization of this Monte Carlo approach to optimization problems is straight forward:

- The current state of the thermodynamic system is analogous to the current solution to the optimization problem
- The energy equation for the thermodynamic system is analogous to the objective function F, and
- The ground state at Temp = 0 is analogous to the global minimum of F.

The major difficulty (art) in implementation of a simulated annealing algorithm is that there is no obvious analogy for the temperature Temp with respect to a free parameter in the optimization problem. Furthermore, avoidance of entrainment in local minima (quenching) is dependent on the "annealing schedule", that is, the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds (after Gray *et al*., 1997). Ideally, when local optimization methods are trapped in a poor local minimum, simulated annealing can 'climb' out.

The algorithm applied in SimEnv is a very fast simulated re-annealing method, named Adaptive Simulated Annealing ASA (Ingber 2004, Ingber 1989 and Ingber 1996). For the above stated probability p the term k_B * Temp is chosen as

$$k_B * Temp = Temp_0 * \exp(-c*t_a^{1/m})$$

where t_a is the annealing time.

The ASA schedule is much faster than Boltzmann annealing, where k_B * Temp = $Temp_0$ / ln(t_a) and faster than fast Cauchy annealing, where k_B * Temp = $Temp_0$ / t_a. For the ASA method the cost function F over the bounded factor space X_k has to be non-convex.

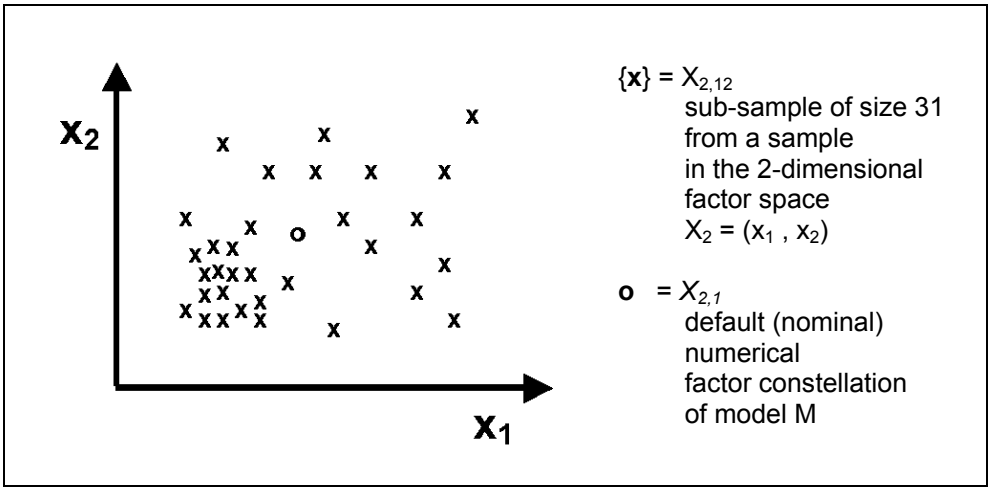


Fig. 4.7 *Part of a sample for an optimization experiment, generated during the experiment*

5 Model Interface

To use any model within SimEnv it has to be interfaced to the simulation environment. SimEnv offers easy coupling techniques at programming language and shell script level. While at language level SimEnv function calls have to be implemented into model source code to address and modify numerically experiment factors, i. e. model parameters, initial or boundary values or drivers of the current single run out of the run ensemble and to output simulation results, at the shell script level communication between the simulation environment and the model can be based on operating system information exchange methods. To plug the model into the simulation environment the variables of the model to be output during experiment performance and to be potentially processed during experiment post-processing have to be declared in the model output description file <model>.mdf. Additionally, the model itself has to be wrapped into a shell script <model>.run.
Model interfacing is related to transferring sampled numerical values of model factors under investigation from the simulation environment to the model and to transferring model output variables under investigation from the model to the simulation environment for later experiment post-processing. Interfacing is supported at the programming language level for C/C++, Fortran, Python, Java, Matlab, Mathematica and GAMS programming languages, the model is implemented in and at shell script level.

5.1 General Approach

SimEnv model interface has to supply a link between the simulation environment and the model and has to address two aspects:
For each single run from the run ensemble

- All experiment factors as defined in the experiment description file <model>.edf (cf. Section 6.1) have to be associated to the corresponding model code entities (parameters, initial or boundary values, drivers). These entities are modified numerically in the model by the sampled values and the default values of the factors according to the specified factor adjustment types. The process of such a modification is call an adjustment. The factor adjustment type specifies how to interfere the current sampled value with the the default value of the entity (cf. Section 6.1).
- All model output variables as defined in the model output description file <model>.mdf (cf. Section 5.3) have to be associated to the corresponding model entities (in general, model state variables) and these entities have to be output to SimEnv data structures during the performance of the model.

Imlementation of this general approach is based on minimal source code manipulation of the model. SimEnv supplies a library with a set of simple functions to interface the model to the simulation environment. Generally speaking,

- Every experiment factor and
- Every model output variable

demand one additional SimEnv function call in the model source code. According to Tab. 5.1 model interface functions are generic.

The function simenv_slice_<lng> announces output of a slice of the data of a defined model output variable. This is good for models with multi-dimensional variables where at least one dimension is omitted in the state variable declaration in the model the source code because the dynamics for this dimension is calculated in place (e.g., time). The assigned variable then has a lower dimensionality than the corresponding variable in the model output description file. Nevertheless, the simenv_slice_<lng>-function ensures that model output over the omitted dimension can be handled in experiment post-processing in common.

Fig. 5.1 shows the conceptual scheme for the SimEnv interface for a Fortran model.

Tab. 5.1 *Generic SimEnv interface functions*
(for language <lng> cf. Tab. 5.2)

Function name	Description
simenv_ini_<lng>	open model coupling interface
simenv_get_<lng>	associate a model source code entity (parameter / initial value / boundary value / driver) with an experiment factor from <model>.edf and assign the adjusted factor value to the entity.
simenv_get_run_<lng>	get the current single run number of the run ensemble
simenv_put_<lng>	associate a model source code entity with a model output variable from <model>.mdf and output it to SimEnv data structures
simenv_slice_<lng>	enable slicing, i.e., a repetitively partial output of model output variables.
simenv_end_<lng>	close model coupling interface

The alignment of the contents of the SimEnv description files and the used SimEnv model interface functions in the model source code is dominated by the description files: These files determine the experiment and the model source code is expected to be well adapted. Nevertheless, this approach is implemented in a flexible manner:

- Function calls in the source code where an experiment factor from <model>.edf and/or a model output variable from <model>.mdf is not associated with are handled during the model performance in such a way that the factors are unadjusted and/or the model output variable is not output. This enables adaption of the model source code for a number of potential experiment factors and model outputs where only a subset of these factors is under consideration in special experiments and/or requested for model output.
- *Vice versa*, model entities that are requested by the corresponding experiment description file as a factor and/or model output description file for model output and where the corresponding SimEnv functions in the model source code are missing are identified as such.

A regular matching between the model output description file and the used SimEnv interface functions in the model source code as well as the above exceptions are reported to the interface log-file <model>.mlog (cf. Tab. 10.8).

Native model output does not influence performance of the model in SimEnv and there is no necessity to disable this output for SimEnv. The user only has to ensure that for an experiment control by the load leveler LoadL the outputs of different single runs do not conflict with each other. Normally, this can be ensured by performing each single run in a special run-related sub-directory (cf. Example 15.10). Native user model output to the terminal is redirected during the experiment to the log-file <model>.nlog.

For running an interfaced model outside SimEnv there are dummy SimEnv libraries to link / run the model with. They ensure the same model dynamics as before interfacing the model to SimEnv (cf. Section 5.12).

Currently, there are SimEnv interfaces for Fortran, C/C++, Python, Java, Matlab, Mathematica and GAMS models. Additionally, there is an interface implementation at shell script leveland for ASCII files. Mixed language models as well as distributed models (cf. Section 5.11) can be run with SimEnv.

Tab. 5.2 *Language suffices for SimEnv interface functions*
(for the Mathematica interface check Section 5.6,
for the GAMS interface check Section 5.7)

<lng>	for model source code
c	C/C++
f	Fortran
py	Python
ja	Java
m	Matlab
as	ASCII file

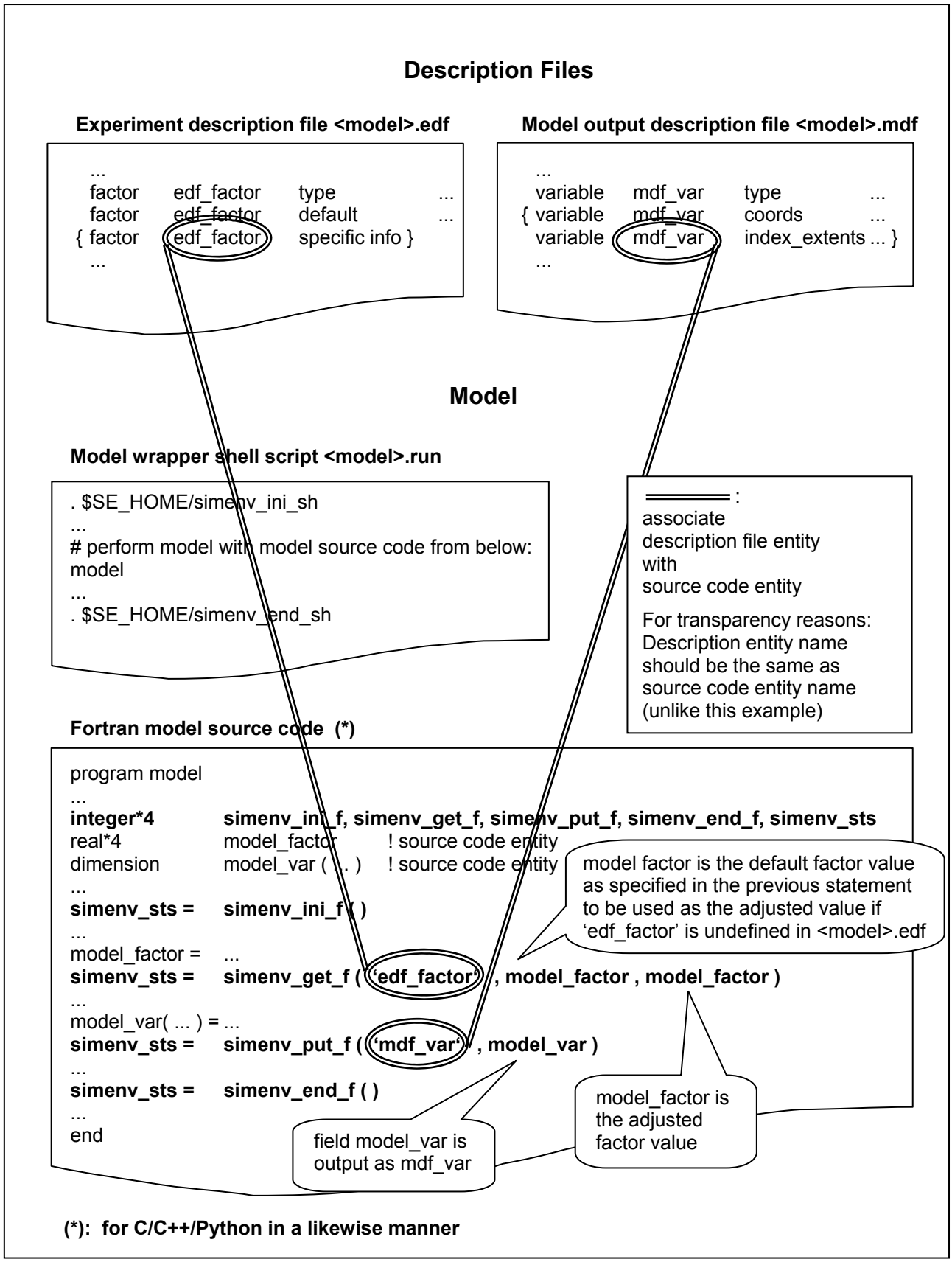


Fig. 5.1 *Conceptual scheme of the model interface for C/C++, Fortran, Python, Java and Matlab*

5.2 Coordinate and Grid Assignments to Variables

To each variable

- Dimensionality **dim(variable)**
- Extents **ext(variable,i)** with i=1 ,..., dim(variable)
- Coordinates **coord(variable,i)** with i=1 ,..., dim(variable)

are assigned to. The dimensionality is the number of dimensions, an extent is related to each dimension and represents the number of elements in that dimension. Extents are always greater than 1. To each dimension a coordinate is assigned to. Coordinates have a name and from all coordinate values the coordinate is defined for a subset is assigned to the extent of the dimension of the variable. Variables of dimensionality 0 do not have a coordinate assignment.
A variable of dimensionality n corresponds to an n-dimensional array, a variable of dimensionality 0 is a scalar.

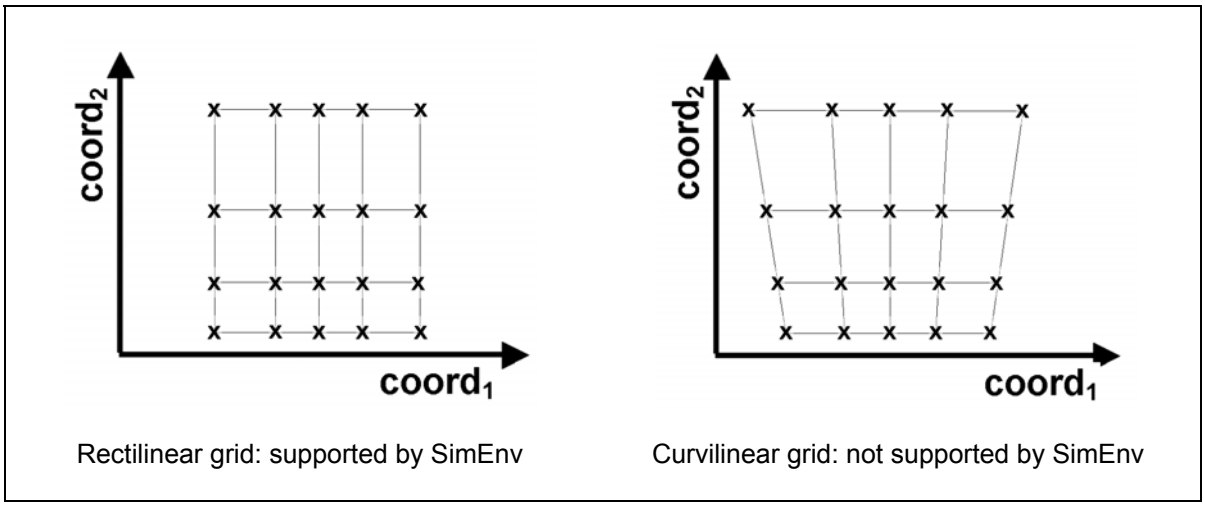


Fig. 5.2 *Grid types*

Additionally, coordinate axes are defined. Each coordinate axis a strictly monotonic sequence of coordinate values, a description and a unit is assigned to. For reasons of simplification in experiment post-processing coordinate axes are assumed as curvilinear.
Each dimension of a variable with a dimensionality > 0 a complete coordinate axis or a part of a coordinate axis is assigned to. Consequently, each variable with a dimensionality > 0 is defined on a coordinate system formed from the assigned coordinates. For reasons of simplification in result evaluation with visualization techniques coordinate systems are assumed as rectilinear (orthogonal with variable distances between adjacent coordinate values). The model output variable values then exist on the grid, spanned up from the coordinate values of the coordinate axes (cf. Fig. 5.2).

Since coordinate axes can be assigned to model output variable dimensions in a flexible manner, model output variables can exist on the same coordinate system or completely or partially disjoint coordinate systems.

5.3 Model Output Description File <model>.mdf

In the model output description file <model>.mdf the model output variables are declared that are to be output by a SimEnv model coupling interface function in the model (code) and are to be post-processed after experiment performance. Additionally, coordinate axes are defined and flexibly assigned to model output variables. Consequently, a model output variable always is defined on a coordinate system, formed from the assigned coordinates to the variable.

Tab. 5.3 *Elements of a model output description file <model>.mdf*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	model output description
coordinate	<co_name>	descr	o	1	<string>	coordinate axis description
		unit	o	1	<string>	coordinate axis unit
		values	m	1	<val_list>	strictly monotonic sequence of coordinate values <co_vals> (for syntax see Tab. 11.6)
variable	<variable_ name>	descr	o	1	<string>	variable description
		unit	o	1	<string>	variable unit
		type	m	1	see Tab. 5.4	variable type in the simulation model
		coords	c1	1	<co_name_1> ,..., <co_name_n>	assigns a coordinate axis by its name to each dimension of the variable. Determines in this way implicitly the dimensionality n of the variable.
		coord_extents	c2	1	<co_val_{11}>: <co_val_{12}> ,..., <co_val_{n1}>: <co_val_{n2}>	assigns start and end coordinate real values from each coordinate axis to the variable. If missing all coordinate values will be used from all assigned coordinates.
		index_extents	c1	1	<in_val_{11}>: <in_val_{12}> ,..., <in_val_{n1}>: <in_val_{n2}>	assigns integer value start and end indices for each dimension to the variable. Indices can be used to address the variable during experiment post-processing.

Each model output variable has a name, a dimensionality and assigned extents, a data type, a description and a unit. The name should correspond to the name of the variable in the simulation model code. Association between these two names is achieved by the SimEnv model interface function simenv_put_* (see below).

<model>.mdf is an ASCII file that holds this information. It follows the coding rules in Section 11.1 on page 141 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

To Tab. 5.3 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- Coordinate and variable names must differ from factor names in experiment description (cf. Section 6.1) and from built-in and user-defined operator names for experiment post-processing (cf. Section 8.5.4).
- Assignment of coordinate axes to variable dimensions and consequently of a grid to a variable is only valid for experiment post-processing. Normally, the simulation model itself will also exploit the same grid structure. Nevertheless, the grid structures of the model are defined autonomously in the model in an explicit or implicit manner and do only correspond to the grid structure in the model output description file symbolically.
- Model output variables with dimensionality 0 are not assigned to a coordinate axis.
- The values of a coordinate have to be ordered in a strictly monotonic sequence. They may be non-equidistant and may be ordered in a decreasing sequence.
- With the sub-keyword '**coord_extents**' only a portion of coordinate values of a coordinate axis can be assigned to a dimension of a variable. This portion is addressed by its begin and end value <co_val_{i1}> and/or <co_val_{i2}>. The number of coordinates values of the portion has to be greater than 1.

<co_val$_{i1}$> > <co_val$_{i2}$> for strictly increasing values of coordinates
<co_val$_{i1}$> < <co_val$_{i2}$> for strictly decreasing values of coordinates

- With the sub-keyword '**index_extents**' portions of variables are made addressable during SimEnv experiment post-processing. In the same way multi-dimensional variables are equipped with indices in the simulation model they also have an index description in the model output description file for purposes of experiment post-processing. It is advisable, that these two descriptions coincide. The index range is described by a start and an end integer value index <in_val$_{i1}$> and/or <in_val_ext$_{i2}$>.
  The index set is a strictly increasing, equidistant set of integer values with an index increment of 1,
  <in_val$_{i1}$> < <in_val$_{i2}$> ,
  <in_val$_{i1}$> ≤ 0 is possible.
- Coordinate values <co_val> and index values <in_val> are assigned in a one-to-one manner.
- For multi-dimensional variables that do not exist on an assigned grid completely or partially, simply assign formal coordinate axes to.
- Specify at least one model output variable in <model>.mdf.

Tab. 5.4 *SimEnv data types*

SimEnv data type (synonyms)		Description		Restriction
byte	int*1	1 byte	integer	not for Python and Java models
short	int*2	2 bytes	integer	not for Python and Java models
int	int*4	4 bytes	integer	
float	real*4	4 bytes	real	
double	real*8	8 bytes	real	not for Python and Java models

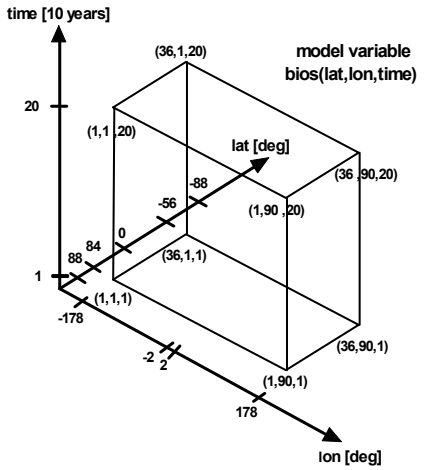
For the following Example 5.1 of a model output description file and the assigned grids for model output variables check Example 1.1 on page 6:

```
general                 descr          World with a resolution of
general                 descr          4° lat    x 4° lon x
general                 descr          4 levels  x 20 time steps
general                 descr          Data centred per lat-lon cell
general                 descr          This file is valid for all models
general                 descr          world_[ f | c | cpp | py | ja |
                                               m | sh | as]

coordinate   lat        descr          geographic latitude
coordinate   lat        unit           deg
coordinate   lat        values         equidist_end 88(-4)-88

coordinate   lon        descr          geographic longitude
coordinate   lon        unit           deg
coordinate   lon        values         equidist_end -178(4)178

coordinate   level      descr          atmospheric vertical level
coordinate   level      unit           level no
coordinate   level      values         list 1,7,11,16

coordinate   time       descr          time in decades
coordinate   time       unit           10 years
coordinate   time       values         equidist_nmb 1(1)20
```

```
variable     atmo      descr          aggregated atmospheric state
variable     atmo      unit           without
variable     atmo      type           float
variable     atmo      coords         lat  , lon  , level , time
variable     atmo      index_extents  1:45 , 1:90 , 1:4   , 1:20

variable     bios      descr          aggregated biospheric state
variable     bios      unit           g/m2
variable     bios      type           float
variable     bios      coords         lat      , lon        , time
variable     bios      coord_extents  84.:-56. , -178.:178. , 1:20
variable     bios      index_extents  1:36     , 1:90       , 1:20

variable     atmo_g    type           int
variable     atmo_g    coords         time
variable     atmo_g    index_extents  1:20

variable     bios_g    type           int
```

Example-file: world_[f | c | cpp | py | ja | m | sh | as].mdf

Example 5.1 *Model output description file <model>.mdf*



Definition of model output variable bios refers to Example 5.1 above.
The triples at the edges of the grid are the indices of model output variable bios(lat,lon,time) for the appropriate grid cells.

Fig. 5.3 *Model output variable definition: Grid assignment*

5.4 Model Interface for Fortran and C/C++ Models

Tab. 5.5 describes the model interface functions that can be used in user models written in Fortran or C/C++ (postfix f for Fortran, c for C/C++)

- to get sampled values of the experiment factors and to adjust them numerically by the factor default value for the current single run of the run ensemble and
- to output model results from the current single run.

In this table the input and output data types are documented for functions used in Fortran. For C/C++ the corresponding data types are valid.
All functions have a 4-byte integer function value (integer*4 and/or int). Implementation of the functions for C/C++ is based on a call by reference for the function arguments.

Tab. 5.5 *Model interface functions for Fortran and C/C++ models*

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ ini_[f | c] ( )	initialize model coupling interface **Perform always as the first SimEnv function in the model. Check the semi-automated model interface for alternatives**	integer*4 simenv_ini_ [f | c] (function value)	return code = 0 ok = 2 I/O error for model output file = 3 error memory allocation = 4 I/O error for simenv_edf_bin.tmp as the structed representation of <model>.edf = 5 I/O error for simenv_mdf_bin.tmp as the structed representation of <model>.mdf = 6 I/O error for <model>.smp = 7 wrong single run number
simenv_ get_[f | c] (factor_name, factor_def_val, factor_adj_val)	get the resulting adjusted value for the factor to be experimented with in the current single run	character*(*) factor_name (input)	name of the factor in <model>.edf
		real*4 factor_def_val (input)	default (nominal) factor value as specified in <model>.edf. If factor_name is not defined in <model>.edf then factor_adj_val is set to factor_def_val
		real*4 factor_adj_val (output)	adjusted factor value
		integer*4 simenv_get_ [f | c] (function value)	return code = 0 ok = 1 factor_name undefined: factor_adj_val := factor_def_val
simenv_ get_run_[f | c] (simenv_run_int, simenv_ run_char)	get run number of the current run as an integer value and a character string	integer*4 simenv_run_int (output)	current run number
		character*6 simenv_ run_char (output)	current run number with leading zeros
		integer*4 simenv_get_run _[f | c] (function value)	return code = 0 ok
simenv_ put_[f | c] (var_name, field)	output model results to native SimEnv output file(s)	character*(*) var_name (input)	name of the variable in <model>.mdf to be output
		dimension field(...), type according to <model>.mdf (input)	data of variable var_name to be stored as simulation results
		integer*4 simenv_put_ [f | c] (function value)	return code = 0 ok = 1 var_name undefined = 2 I/O error for model output file

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ slice_[f | c] (var_name, idim, ifrom, ito)	announce to output at the next corresponding simenv_put_[f | c] call only a slice of variable var_name. This announcement becomes inactive after performance of the corresponding simenv_put_[f | c]	character*(*) var_name (input)	name of the variable in <model>.mdf to be sliced
		integer*4 idim (input)	dimension to be sliced
		integer*4 ifrom (input)	slice to start at position ifrom. ifrom corresponds to an index from index_extents in <model>.mdf
		integer*4 ito (input)	slice to end at position ito. ito corresponds to an index from index_extents in <model>.mdf
		integer*4 simenv_slice_ [f | c] (function value)	return code = 0 ok = 1 var_name undefined = 3 inconsistency between variable and idim, ifrom, ito = 4 slice storage exceeded = 5 warning: slice overwritten or slice represents the total dimension
simenv_ end_[f | c] ( )	close model coupling interface **Perform always the last SimEnv function in the model**	integer*4 simenv_end_ [f | c] (function value)	return code = 0 ok = 2 I/O error for model output file

- Make sure consistency of type and dimension declarations between the model output variables in model source code and the corresponding variable declarations in the model output description file <model>.mdf.
- Model output variables that are not output completely or only partially within the user model are handled in experiment post-processing as their corresponding nodata-values (cf. Tab. 10.13).
- Application of simenv_slice_[f | c] demands a corresponding slice entry in the configuration file <model>.cfg. For more information check Section 10.1.
- Application of simenv_slice_[f | c] for NetCDF model output may result in a higher consumption of computing time for each single run of the experiment compared with NetCDF model output without simenv_slice_*. For this case, keep in mind the trade-off between the demand for computing time and the demand for main memory.
- The include file simenv_mod_[f | c].inc from the sub-directory inc of the SimEnv home directory can be used in a model to declare the SimEnv model interface functions as integer*4 / int for Fortran / C/C++.
- Apply the shell script
  simenv_mod_[f | c | cpp].lnk <model_name>
  from the SimEnv library directory $SE_HOME/lib to compile and link an interfaced model
- User models implemented in C/C++ or Fortran have to be linked with the following libraries to interface them to the simulation environment
  - $SE_HOME/lib/libsimenv.a
  - libnetcdf.a from /usr/local/lib or /usr/lib
- Tab. 15.12 lists the additionally used symbols when interfacing a Fortran or C/C++ model to SimEnv.
- In
  - Example 15.1 on page 167 the model world_f.f
  - Example 15.3 on page 170 the model world_c.c
  - Example 15.4 on page 172 the model world_cpp.cpp

  are explained.

5.5 Model Interface for Python, Java and Matlab Models

Due to the special features of Python, Java and Matlab, the model interface components for simenv_get und simenv_get_run differs from that for Fortran and C/C++ in Section 5.4. Additionally, the model interface for Python and Java does not support all data types (cf. Tab. 5.4). Tab. 5.6 summarizes the model interface modules for a Python and Java models.

Tab. 5.6 *Model interface modules / methods / functions for Python, Java and Matlab models (addressed as "functions")*

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ ini_[py | ja | m] ( )	initialize model coupling interface **Perform always as the first SimEnv function in the model. Check the semi-automated model interface for alternatives**	string (py) / int (ja / m) simenv_ini_* (function value)	return code = 0 ok
simenv_ get_[py | ja | m] (factor_name, factor_def_val))	get the resulting adjusted value for the factor to be experimented with in the current single run	string factor_name (input)	name of the factor in <model>.edf
		float factor_def_val (input)	default (nominal) factor value as specified in <model>.edf. If factor_name is not defined in <model>.edf then factor_adj_val is set to factor_def_val
		float simenv_get_* (function value)	adjusted factor value factor_adj_val If an error occurred then function value =-999.99
simenv_ get_run_ [py | ja | m] ( )	get the run number of the current run as a string	string simenv_get_run_* (function value)	current run number as string of the length 6 with leading zeros. If an error occurred then function value = '------'
simenv_ put_[py ja | m] (var_name, field)	output model results to native SimEnv output file(s)	string var_name (input)	name of the variable in <model>.mdf to be output
		declaration of field(...) according to <model>.mdf (input)	data of variable var_name to be stored as simulation results
		string (py) / int (ja / m) simenv_put_* (function value)	return code = 0 ok ≠ 0 an error occured

Function name	Function description	Arguments / function value	Argument / function value description
simenv_ slice_ [py | ja | m] (var_name, idim, ifrom, ito)	announce to output at the next corresponding simenv_put_* call only a slice of variable var_name. This announcement becomes inactive after performance of the corresponding simenv_put_*	string var_name (input)	name of the variable in <model>.mdf to be sliced
		int idim (input)	dimension to be sliced
		int ifrom (input)	slice to start at position ifrom. ifrom corresponds to an index from index_extents in <model>.mdf
		int ito (input)	slice to end at position ito. ito corresponds to an index from index_extents in <model>.mdf
		int simenv_slice_* (function value)	return code = 0 ok ≠ 0 an error occured
simenv_ end_ [py | ja | m] ( )	close model coupling interface **Perform always as the last SimEnv function in the model**	string (py) / int (ja / m) simenv_end_* (function value)	return code = 0 ok

- Make sure consistency of type and dimension declarations between the model output variables in model source code and the corresponding variable declarations in the model output description file <model>.mdf.
- Model output variables that are not output completely or only partially within the user model are handled in experiment post-processing as their corresponding nodata-values (cf. Tab. 10.13).
- Application of simenv_slice_[py | ja | m] demands a corresponding slice entry in the configuration file <model>.cfg. For more information check Section 10.1.
- Application of simenv_slice_[py | ja | m] results in a higher consumption of computing time for each single run of the experiment compared without simenv_slice_*. For this case, keep in mind the trade-off between the demand for computing time and the demand for main memory.
- Application of simenv_slice_[py | ja | m] for NetCDF model output may result in a higher consumption of computing time for each single run of the experiment compared with NetCDF model output without simenv_slice_*. For this case, keep in mind the trade-off between the demand for computing time and the demand for main memory.
- SimEnv Python model interface modules are declared in the file simenv.py in the sub-directory bin of the SimEnv home directory. To use these modules in a Python model import it by
  from simenv import *
  and refer to them for example by
  simenv_get_py.
- SimEnv Java model interface methods are declared in the file Simenv.java in the sub-directory bin of the SimEnv home directory. The corresponding class file Simenv.class is also located there. Specify in <model>.run or in the .profile file the CLASSPATH by
  export CLASSPATH=./:$SE_HOME/bin:$CLASSPATH
  before calling java to run the model.
  To use an interface method in a Java model refer it for example by
  Simenv.simenv_get_py.
- SimEnv Matlab model interface functions are in the sub-directory bin of the SimEnv home directory. Insert into the file $HOME/matlab/startup.m
  addpath ([getenv('SE_HOME') '/bin']);
  before performing an experiment with a Matlab model.

Start a Matlab model in <model>.run by

matlab –nojvm –nosplash < model_name

Contrary to general Matlab syntax variable and factor names as the first argument in simenv_get_m, simenv_slice_m and simenv_put_m are not case sensitive and are transformed to lower cases in the appropriate Matlab interface function. See also Section 10.6.

- Errors that occur during performance of one of the above Python, Java or Matlab interface modules / methods are directly reported to the log-file <model>.nlog.

Set in $HOME/.profile the Python, Java and/or Matlab environment: include the path to python, Java and/or Matlab in the PATH environment variable and specify for Python the PYTHONPATH environment variable accordingly to the need of the Python model. For more information check Section 10.8.

In Example 15.5 on page 173 the Python model world_py.py is described in detail, in Example 15.6 on page 174 the Java model world_ja.java and in Example 15.7 on page 175 the Matlab model world_m.m.

5.5.1 Standard Dot Scripts for Python, Java and Matlab Models

<model>.ini

<model>.ini (cf. Section 7.1 on page 67) is for Python, Java and Matlab models a mandatory shell script and has to have the same contents for all Python, Java and/or Matlab models:

```
. $SE_HOME/bin/simenv_ini_[ py | ja | m ]
# return code from simenv_ini_[ py | ja | m ] is rc_simenv_ini_[ py | ja | m ] (=0: ok, =1: error)

# additional user-model specific commands can be implemented up from here
if test $rc_simenv_ini_[ py | ja | m ] = 0
then
    ...
fi

# return always with the return code rc_simenv_ini_[ py | ja | m ]
exit $rc_simenv_ini_[ py | ja | m ]
```

For an experiment restart with a Python, Java or Matlab model (cf. Section 7.4 on page 73) <model>.ini has to be performed again. To force this specify in <model>.cfg (cf. Section 10.1 on page 125) for the sub-keyword ‘restart_ini’ the value “yes”.

5.6 Model Interface for Mathematica Models

For Mathematica models a simple interface to SimEnv is implemented. It is based on

- generating automatically per single run a temporary Mathematica model by prefixing the original model with a piece of Mathematica model code that is generated automatically by SimEnv.
- performing this temporary model
- transferring the model output from external files to SimEnv model output structures.

Set in the file $HOME/.profile the Mathematica environment: include the path to MathKernel in the PATH environment variable. For more information check Section 10.8.

simenv_get function

The generic simenv_get function for a Mathematica model and running the model is performed by the SimEnv dot script

```
. $SE_HOME/bin/simenv_run_mathematica
```

This dot script has to be called in <model>.run. It expects that the Mathematica model has the name <model>.m where <model> is the model name the service is started with.

To enable the adjustment of a factor <factor_name> in the model during the performance of any single run it is necessary to modify the model source code with respect to the initial settings of the factors. Let depend the assignment of the default value <factor_def_val> to the factor variable <factor_name> in the model source code whether this variable was already set to its adjusted value by:

```
if [ ValueQ[<factor_name>] == False ,
     <factor_name> = <factor_def_val> ,
     <factor_name> = <factor_name> ];
```

For an experiment with k factors the temporary Mathematica model for single run number <simenv_run_int> has the following structure:

<factor_name$_1$> = <factor_value$_{1<simenv_run_int>}$> ;
…
<factor_name$_k$> = <factor_value$_{k<simenv_run_int>}$> ;
<model>

This file is generated in a temporary sub-directory run<simenv_run_char> of the current workspace. The sub-directory itself is created automatically when performing the single run <simenv_run_int>.

Store the Mathematica model in the current workspace the SimEnv simulation service simenv.[run | rst] is started from.

Since the original model is prefixed by the above piece of code that defines the adjustments of the factors, all statements (e.g., “clearall”) that clear the model variables have to be deleted from the original model source code.

simenv_put function

For the Mathematica model interface a dedicated simenv_put function does not exist. SimEnv expects the Mathematica model to write model output to external files. These files can be transferred into SimEnv model output by writing a specific simenv_put_sh executable (cf. Section 5.8) or for ASCII model output files by applying simenv_put_as[_simple] (cf. Section 5.9). Both interfaces have to be incorporated into <model>.run.

<model>.edf
While normally for the model interface the names of corresponding factors in the model description file <model>.edf and the model source code can differ and are associated by the first argument of the interface function simenv_put_* (cf. Fig. 5.1) the names have to coincide for the Mathematica model interface. Since in Mathematica variables are case sensitive they have to be declared in the experiment description file <model>.edf also in a case sensitive manner.

An example for <model>.run can be found in Example 15.8.

5.7 Model Interface for GAMS Models

SimEnv allows to interface GAMS models to the experiment shell. A GAMS (main) model interfaced to SimEnv can call GAMS sub-models. SimEnv expects that the GAMS main model

- is located in the file <model>.gms where <model> is the model name a SimEnv service is started with.
- and all optional GAMS sub-models are stored in the current workspace the SimEnv services simenv.[chk | run | rst] are started from.

Therefore, two additional include-statements have to be inserted into these GAMS model source code files where experiment factors are to be adjusted or model variables are to be output to SimEnv. GAMS model source code files to be interfaced to SimEnv are one GAMS main model and optionally a number of GAMS sub-models that are called directly from the GAMS main model. Additional GAMS sub-programs (included files) are not affected bei SimEnv, but one should keep in mind that the GAMS code within SimEnv will be executed in a sub-directory of the current workspace (see below) and so the include statements have to be changed, if the files are addressed in a relative manner (see below).

- The include files are
  <model>_simenv_get.inc
  <model>_simenv_put.inc
- During experiment preparation the file <model>_simenv_put.inc and during experiment performance files <model>_simenv_get.inc are generated automatically to forward GAMS model output to SimEnv data structures and to adjust investigated experiment factors, respectively.
  These include files correspond to the simenv_put and simenv_get model interface functions at the language level (cf. Sections 5.4 and 5.5).
- The GAMS include statement $include <model>_simenv_get.inc has to be placed in the GAMS model file at such a position where all the GAMS variables are declared. Directly before the include statement the factor default values have to be assigned to factor variables, that are introduced additionally in the model. Directly after the include statement the factor variables with the adjusted factor values have to be assigned to the model variables.
- The GAMS include statement $include <model>_simenv_put.inc has to be placed in the GAMS model file at such a position where all the variables from the model output description file can be output by GAMS put-statements.
- In the course of experiment preparation the GAMS model and all sub-models that are specified in <model>.gdf (see below) are transformed automatically. Each GAMS model single run from the run ensemble is performed in a separate sub-directory run<simenv_run_char> of the current workspace. The sub-directories are created automatically. Transformed GAMS models and sub-models are copied to this sub-directory and are performed from there. Keep this in mind when specifying in any GAMS model include statements with relative paths.

In Example 15.9 on page 178 the model gams_model.gms is described in detail.

Note the following information:

- To output the GAMS model status to SimEnv a
  ```
  PARAMETER modstat
  ```
  has to be declared and the statement
  ```
  modstat = <model_name>.modelstat
  ```
  has to be incorporated in the GAMS model above the $include <model>_simenv_put.inc line. The variable modstat has to be stated in the model output description file <model>.mdf and the GAMS description file <model>.gdf.

Set in the file $HOME/.profile the GAMS environment: include the path to gams in the PATH environment variable. For more information check Section 10.8.

5.7.1 Standard Dot Scripts for GAMS Models

<model>.ini
<model>.ini (cf. Section 7.1 on page 67) is for GAMS models a mandatory shell script and has to have the contents for all GAMS models:

```
. $SE_HOME/bin/simenv_ini_gams
# return code from simenv_ini_gams is rc_simenv_ini_gams (=0: ok, =1: error)

# additional user-model specific commands can be implemented up from here
if test $rc_simenv_ini_gams = 0
then
    ...
fi

# return always with the return code rc_simenv_ini_gams
exit $rc_simenv_ini_gams
```

For an experiment restart with a GAMS model (cf. Section 7.4 on page 73) <model>.ini has to be performed again. To force this, specify in <model>.cfg (cf. Section 10.1 on page 125) for the sub-keyword 'restart_ini' the value "yes".

<model>.run
<model>.run (cf. Section 7.1 on page 67) has for each GAMS model the same contents:

```
#! /bin/sh
. $SE_HOME/bin/simenv_ini_sh
. $SE_HOME/bin/simenv_run_gams
. $SE_HOME/bin/simenv_end_sh
```

<model>.end
<model>.end (cf. Section 7.1 on page 67) is for GAMS models a mandatory shell script and has to have the contents for all GAMS models:

```
. $SE_HOME/bin/simenv_end_gams

# additional user-model specific commands can follow
```

Python programming language is used to prepare, run and to end an experiment with a GAMS model.

5.7.2 GAMS Description File <model>.gdf, <model>.edf, <model>.mdf

- An ASCII GAMS description file **<model>.gdf** (see below) has to be supplied

The ASCII GAMS description file <model>.gdf is intented to specify the GAMS sub-models and assigned factors and model output variables in detail. Derived from this information a block of lines for each GAMS sub-model with a simenv_get.inc file and/or a simenv_put.inc file is created. The file <model>.gdf holds the specific characteristics of GAMS model input and output needed by SimEnv to generate GAMS put-statements. All model output variables from the model output description file and all factors from the factor description file have to be used in this file again.

<model>.gdf is an ASCII file that follows the coding rules in Section 11.1 on page 141 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

Tab. 5.7 *Elements of a GAMS description file <model>.gdf*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	GAMS coupling description
		keep_runs	o	1	<val_list>	value list of run numbers where single GAMS model runs are to be stored by keeping their corresponding sub-directories (for syntax see Tab. 11.6)
		time_limit	o	1	<int_val>	wall clock limit in seconds for each GAMS model single run
		options	o	1	<string>	string of options, GAMS main model is started with from command line
model	<model_name> (without extension .gms)	descr	o	1	<string>	(sub-)model output description
		type	m	1	[main | sub]	identifies GAMS main or sub-model
		get	m	exactly number of factors	<factor_name>	get resulting adjustment for <factor_name> to this model
		put	m	exactly number of model output variables	(<var_name>{.<suffix_set>}{(<index_set>)}){<format>}	put values of SimEnv model output variable <var_name> from this model to SimEnv output. GAMS variable <var_name> has the specified suffix and index sets and is interfaced from GAMS to SimEnv according to <format>

To Tab. 5.7 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- Each factor and each model output variable as declared in <model>.edf and <model>.mdf respectively has to be used in the value-field of <model>.gdf exactly one time.
- To each GAMS model <model_name> an arbitrary number of factors and model output variables can be assigned to by the corresponding sub-keyword ‘get’ and/or ‘put’.
  To each sub-model (‘type’ = “sub”) at least one ‘get’ or one ‘put’ sub-keyword must be assigned to. The main model (‘type’ = “main”) can be configured without any sub-keyword ‘get’ and ‘put’. This is useful when the main model simply calls sub-models.
- Each model <model_name> in <model>.gdf with at least one sub-keyword ‘get’ has to have an $include <model_name>_simenv_get.inc statement in the corresponding GAMS model file <model_name>.gms
- Each model <model_name> in <model>.gdf with at least one sub-keyword ‘put’ has to have an $include <model_name>_simenv_put.inc statement in the corresponding GAMS model file <model_name>.gms
- There has to be exactly one main GAMS model, identified by the sub-keyword ‘type’ value “main”. All other models have to be of sub-keyword type value “sub”.
- The value-field for the sub-keyword ‘put’ is adapted to GAMS syntax to output GAMS model output variables. Afterwards this output is used to generate the appropriate SimEnv output.

<index_set> is mandatory for variables with a dimensionality > 0. Otherwise, specification of <index_set> is forbidden. Indices as used in the GAMS model are separated from each other by comma.

- The sub-keyword 'time_limit' enables limitation of each GAMS model single run in the run ensemble to a maximum wall clock time consumption. If this threshold is reached the single run is aborted and the following single run started. In general, SimEnv nodata values will be assigned to the results of the aborted single runs. The sub-keyword 'time_limit' can be necessary since each GAMS model single run itself is an optimization procedure which could result in an unfeasible wall clock time consumption. If the sub-keyword is not used in the gdf-file wall clock time consumption per single run is unlimited.

With respect to Example 15.9 the GAMS description file could look like

```
general                 descr           GAMS model output description
general                 descr           for the examples in the SimEnv
general                 descr           User Guide
general                 keep_runs       list 0,1

model    gams_model     descr           this is the only GAMS model to use
model    gams_model     type            main
model    gams_model     get             dem_ny
model    gams_model     get             dem_ch
model    gams_model     put             x.l(i,j):10:5
model    gams_model     put             a(i):10:5
model    gams_model     put             z.l
model    gams_model     put             modstat
```

Example file: gams_model.gdf

Example 5.2 *GAMS description file <model>.gdf*

If the model gams_model from the above Example 5.2 would be coupled with two additional GAMS sub-models sub_m1 and sub_m2 where both sub-models interact with SimEnv the GAMS description file could look like
(without taking into consideration plausibility with respect to model contents)

```
model    gams_model     type            main
model    gams_model     put             modstat

model    sub_m1         type            sub
model    sub_m1         get             dem_ny
model    sub_m1         put             x.l(i,j):10:5
model    sub_m1         put             a(i):10:5

model    sub_m2         type            sub
model    sub_m2         get             dem_ch
model    sub_m2         put             z.l
```

or

```
model    gams_model     type            main

model    sub_m1         type            sub
model    sub_m1         get             dem_ny
model    sub_m1         put             x.l(i,j):10:5
model    sub_m1         put             a(i):10:5
```

```
model    sub_m2        type            sub
model    sub_m2        get             dem_ch
model    sub_m2        put             z.l
model    sub_m2        put             modstat
```

Example 5.3 *GAMS description file for coupled GAMS models*

<model>.edf
While for the C/C++/Fortran/Python/Java/Matlab model interface the names of corresponding factors in the model description file <model>.edf and the model source code can differ and are associated by the first argument of the interface function simenv_put_* (cf. Fig. 5.1) the names have to coincide for the GAMS model interface.
In the GAMS model code the factors specified in the experiment description file have to be of type PARAMETER and have be defined before the include statement $include simenv_get.inc.

<model>.mdf
Corresponding variables in the model output description file and in the GAMS model source code must have same names. Variables have to be always of type float in the model output description file. In GAMS model code the model output variables declared in the model output description file can be of the numeric types VARIABLES or PARAMETER. The maximum dimensionality of a SimEnv model output variable used in a GAMS model is restricted to 4. Additionally, each model output variable must not exceed a size of 64 MByte.

With respect to Example 15.9 the model output description file could look like

```
coordinate   plant     descr          canning plants
coordinate   plant     unit           plant number
coordinate   plant     values         equidist_end 1(1)2

coordinate   market    descr          canning markets
coordinate   market    unit           market number
coordinate   market    values         equidist_end 1(1)3

variable     a         descr          plant capacity
variable     a         unit           cases
variable     a         type           float
variable     a         coords         plant
variable     a         index_extents  1:2

variable     x         descr          shipment quantities
variable     x         unit           cases
variable     x         type           float
variable     x         coords         plant , market
variable     x         index_extents  1:2   , 1:3

variable     z         descr          total transportation costs
variable     z         unit           10^3 US$
variable     z         type           float

variable     modstat   descr          model status
variable     modstat   type           float
```

Example file: gams_model.mdf

Example 5.4 *Model output description file for a GAMS model*

5.7.3 Files Created during GAMS Model Performance

Additionally to the files listed in Tab. 10.8, during the performance of a GAMS model the files <gams_model>_[pre | main | post].inc are created temporarily in the current workspace by <model>.ini and are deleted after the whole experiment where <gams_model> is a placeholder for the model of type main and all models of type sub in the gdf-file.

During experiment performance of a GAMS model each single run <simenv_run_int> from the experiment is performed individually in a sub-directory run<simenv_run_char> of the current workspace. Each directory is generated automatically before performing the corresponding single run and removed after perfomance of this single run. With the sub-keyword 'keep_runs' the user can force to keep sub-directories for later check of the transformed model code and its performance.

GAMS main model terminal output is redirected to the log-file main_model<simenv_run_char>.nlog in the corresponding sub-directory run<simenv_run_char>. For re-direction of the terminal output from sub-models and from solvers the modeler is responsible for. It is recommended to call all GAMS sub-models with the GAMS command line option string

```
ll=0 lo=2 lf=<any_name>.nlog dp=0 Optdir=../
```

that re-directs GAMS submodel and solver terminal output to the file <any_name>.nlog in the sub-directory run<simenv_run_char> of the current workspace (cf. Example 15.9). The main model is called by default with

```
ll=0 lo=2 lf=main_model<simenv_run_char>.nlog dp=0 Optdir=../
```

All files with the extension nlog in the sub-directory run<simenv_run_char> are copied to the SimEnv log file <model>.nlog.
For both cases Optdir=../ implies that all GAMS solver option files have to be located in the current working directory. With the options sub-keyword 'options' additional command line options can be specified in <model>.gdf for the main model.

5.8 Model Interface at Shell Script Level

For models that do not allow to implement the model coupling interface at programming language level (e.g., because source code is not available) SimEnv supplies a coupling interface at shell script level by a set of dot scripts: The shell script <model>.run (cf. Section 7.1 on page 67) is used to wrap the model and optionally to have at disposal corresponding functionality of the SimEnv model interface functions of Tab. 5.5. For additional interfaces at the shell script level using ASCII files see Section 5.9.

- For the model interface at the shell script level, i.e., within the shell script <model>.run the adjusted experiment factors for the current single run from the whole run ensemble can be made available to forward them to the model under investigation by any means the modeller is responsible for.
  One common way to forward experiment factors to the model is to place current numerical factor values as arguments to the model executable at the model command line in Unix or Linux. Another way could be to read the factors from a special file in a special file format.
- While for the C/C++/Fortran/Python/Java/Matlab model interface the names of corresponding factors in the model description file <model>.edf and the model source code can differ and are associated by the first argument of the interface function simenv_put_* (cf. Fig. 5.1) the names have to coincide for the model interface at the shell script level.
- Directly before performing the dot script $SE_HOME/bin/simenv_get_sh make sure that the shell script variables factor_name and factor_def_val have been specified. At the end of the dot script simenv_get_sh these variables are set again to empty strings.
- After running the dot script $SE_HOME/bin/simenv_get_sh the name of an experiment factor <factor_name> from the experiment description file <model>.edf is available in <model>.run as a shell script variable <factor_name> and the resulting adjusted value of the factor is available as $<factor_name>.

- After running the model model output has to be identified and potentially transformed within <model>.run for SimEnv output. To do this simply write a model related simenv_put_sh as a transformation program that reads in all the native model output and outputs it to SimEnv by applying the model interface functions simenv_*_* from the SimEnv model interfaces at language level.
- Tab. 10.10 lists the built-in (pre-defined) shell script variables that are defined and/or used by the dot scripts $SE_HOME/bin/simenv_*_sh and are directly available in <model>.run.
- Notice:
  To perform a dot script (cf. Section 15.7 - Glossary) it has to be preceded by a dot and a space.

In Example 15.10 on page 179 the model shell script world_sh.run is described in detail.

Tab. 5.8 *Model interface functions at shell script level*

Dot script name	Command description	Inputs / outputs	Input / output description
$SE_HOME/bin/simenv_ini_sh	initialize current single run **Perform always and as the first SimEnv dot script in <model>.run and <model>.rst. Check the semi-automated model interface for alternatives for <model>.run**	SE_RUN (output)	operating system environment variable SE_RUN is set to the current run number of the simulation experiment
factor_name= '<factor_name>' factor_def_val= <factor_def_val> $SE_HOME/bin/simenv_get_sh	get the resulting adjusted value for the factor to be experimented with in the current single run	script variable factor_name (input)	name of the factor in <model>.edf
		script variable factor_def_val (input)	default (nominal) factor value. If factor_name is not defined in <model>.edf then factor_adj_val is set to <factor_def_val>
		script variable factor_name (output)	shell script variable with the same name as the value of factor_name. Script variable value is the adjusted factor value <factor_adj_val>.
$SE_HOME/bin/simenv_get_run_sh	get the run number of the current run as an integer and a character script variable	simenv_run_char (output)	shell script variable with the current run number with leading zeros
		simenv_run_int (output)	shell script variable (type integer) with the current run number
$SE_HOME/bin/simenv_put_sh	**Not available at shell script level**		**Write a your own model related simenv_put_sh at the language level using the SimEnv model interface functions**
$SE_HOME/bin/simenv_slice_sh	**Not available at shell script level**		
$SE_HOME/bin/simenv_end_sh	wrap up current single run **Perform always and as the last SimEnv dot script in <model>.run and <model>.rst**		

```
. $SE_HOME/bin/simenv_ini_sh

# get adjusted value for the a factor p_def, defined in the edf-file
factor_name='p_def'
factor_def_val=2.
. $SE_HOME/bin/simenv_get_sh
# now shell script variable p_def          is   available
# value of shell script variable p_def     is   according to edf-file

# get adjusted value for a factor p_undef, not defined in edf-file
factor_name='p_undef'
factor_def_val=-999.
. $SE_HOME/bin/simenv_get_sh
# now shell script variable p_undef        is   available
# value of shell script variable p_undef   is   -999.

# ...

. $SE_HOME/bin/simenv_end_sh
```

Example file: world_sh.run

Example 5.5 *Addressing factor names and values for the model interface at shell script level*

5.9 Model Interface for ASCII Files

The SimEnv ASCII interface addresses constellations where

- a model expects factor names and their adjusted values in an ASCII file
- model output is stored to ASCII files.

Tab. 5.9 lists those SimEnv dot scripts and shell scripts that represent the ASCII interface. They have to be applied in the model wrap shell script <model>.run. They can be used together with the interface at the shell script level (cf. Section 5.8).

Tab. 5.9 *Model interface functions at ASCII level*

Dot /shell script name	Command description	Inputs / outputs	Input / output description
$SE_HOME/bin/simenv_get_as	get the names and the resulting adjusted values of all factors to be experimented with in the current single run	ASCII file <model>.as<simenv_run_char> (output)	After performing simenv_get_sh the ASCII file <model>.as<simenv_run_char> contains all factor names and resulting adjusted values in the form <factor_name> <factor_adj_val> Sequence of the factor lines in the file corresponds to the sequence of the factors in the experiment description file <model>.edf
$SE_HOME/bin/simenv_put_as <file_name> { <coord> }	transfer ASCII file to SimEnv model output in safe mode **simenv_put_as is a normal shell script and NOT a dot script**	<file_name> (input)	Name of an ASCII file that is transferred to SimEnv model output according to model output variable coordinate <coord>
		<coord> (input)	Name of a model output variable coordinate. Lines in <file_name> correspond to coordinate values. If <coord> is not specified <file_name> has to be a one-record file.

Dot /shell script name	Command description	Inputs / outputs	Input / output description
$SE_HOME/bin/ simenv_ put_as_simple <file_name> { <coord> }	transfer ASCII file to SimEnv model output in simple mode **simenv_put_as_ simple is a normal shell script and NOT a dot script**	<file_name> (input)	Name of an ASCII file that is transferred to SimEnv model output according to model output variable coordinate <coord>
		<coord> (input)	Name of a model output variable coordinate. Lines in <file_name> correspond to coordinate values. If <coord> is not specified <file_name> has to be a one-record file.

After performing the dot script simenv_get_as an ASCII file <model>.as<simenv_run_char> holds lines with a factor name and its resulting adjusted value per line. Each factor name is separated from its resulting adjusted value by at least one character space.

simenv_put_as and simenv_put_as_simple can be used to transfer ASCII model output to SimEnv model output data structures. These are the only SimEnv scripts that can be used in <model>.run that are shell scripts and not dot scripts. Both they have two arguments. The first argument specifies the ASCII file name that is to be transferred. The second argument is the name of a coordinate as specified in <model>.mdf.

Both shell scripts transfer the ASCII file data to SimEnv model output file as follows:

- A SimEnv model output variable is defined on a rectilinear grid that is composed from coordinates (cf. Section 5.2). By specifying a coordinate name as the second argument all these model variable values are expected in the ASCII file that have this coordinate as their first coordinate (cf. Section 5.3).
- The lines in the ASCII file correspond to the coordinate axis values in that sequence as defined in <model>.mdf.
- The columns in the ASCII file correspond to the variables with the first coordinate as specified in the second argument in that sequence of the variables as defined in <model>.mdf. A multi-dimensional variable occupies a block of contiguous columns. The sequence of all columns of all lines of this variable is according to the Fortran storage model (cf. Section 15.7 - Glossary).
- Variables with the same first coordinate but with different coordinate extents (variable sub-keyword coord_extents in <model>.mdf) have to be harmonised line by line: The set of all lines is the union of all defined coordinate axis values from all variables. To ensure synchronisation across columns, variable values for undefined coordinate values of a variable have to be output to the file as any real*4 / float nodata placeholder <nodata>.
- The values of the ASCII file are interpreted as of type real*4 / float. They are transferred to SimEnv model output according to their defined data type. If a real*4 / float value is outside the definition range of the data type it is set to the SimEnv nodata element of this data type (cf. Tab. 10.13).
- If no coordinate is defined as the second argument the values of all zero-dimensional variables are expected to be in the ASCII file. Consequently, the file can have only one record with data values.
- The shell scripts simenv_put_as and simenv_put_as_simple differ in how to read each line of the ASCII file. simenv_put_as handles the file as an ASCII data file, defined in Section 11.3 with the exception that data files are not limited to 1000 characters. Consequently, a file can have comment and blank lines when transferring by simenv_get_as to SimEnv. Additionally, the number of columns per line is checked and missing columns are added as nodata values. In contrast, simenv_put_as_simple just applies a simple Fortran read statement per expected line without any checking routines. Due to its extensive interpretation efforts simenv_put_as is rather slow. As a rule of thumb simenv_put_as_simple should be used for file with more than 2000 columns where one can trust in the file structure.

Having a model output variable definition as in Example 5.1 on page 31.

```
$SE_HOME/bin/simenv_put_as atmo_g.ascii time
```

Since atmo_g is the only variable with time as the first coordinate the file atmo_g.ascii can only hold this variable. The 1st record of the file with data corresponds with time = 1, the 20th record with data with time = 20. There is only one column.

```
$SE_HOME/bin/simenv_put_as bios.ascii lat
```

Assuming that variable atmo is not defined.
Since bios is the only variable with lat as the first coordinate the file bios.ascii can only hold this variable. The 1st record of the file with data corresponds with lat = 84, the 36th record with data with lat = -56. There are 90*20 = 1800 columns. The file has to hold bios(lat,lon,time) in the following structure, shown are the indices of bios:

column #/ line #	1	2	…	90	91	…	90*20
1	(84,-178,1)	(84,-174,1)	…	(84,178,1)	(84,-178,2)	…	(84,178,20)
…	…	…	…	…	…	…	…
36	(-56,-178,1)	(-56,-174,1)	…	(-56,178,1)	(-56,-178,2)	…	(-56,178,20)

```
$SE_HOME/bin/simenv_put_as_simple atmo_bios.ascii lat
```

atmo and bios are the variable with lat as the first coordinate. According to the sequence in world_as.mdf the file atmo_bios.ascii has to hold in its first columns the variable atmo, followed by the variable bios. Since bios is defined for the coordinate lat only on the subrange 2 – 37 of the complete range 1 – 45 for atmo values with numerical nodata-placeholder <nodata> (e.g., 0.0) have to be set for all values of bios in file records 1 and 38 to 45. The first record of the file corresponds for atmo with lat = 88, for bios <nodata> has to be assigned.. The 45th record corresponds for atmo with lat = -88, for bios <nodata> has to be assigned. There are 90*4*20 + 90*20 = 9000 columns, that's why simenv_put_as_simple is used instead of simenv_put_as. The file has to hold atmo(lat,lon,level,time) and bios(lat,lon,time) in the following structure, shown are the indices:

	atmo			bios		
column #/ line #	1	…	90*4*20	90*4*20+1	…	9000
1	(88,-178,1,1)	…	(88,178,16,20)	<nodata>	…	<nodata>
2	(84,-178,1,1)	…	(84,178,16,20)	(84,-178,1)	…	(84,178,20)
…	…	…	…	…	…	…
37	(-56,-178,1,1)	…	(-56,178,16,20)	(-56,-178,1)	…	(-56,178,20)
38	(-60,-178,1,1)	…	(-60,178,16,20)	<nodata>	…	<nodata>
…	…	…	…	…	…	…
45	(-88,-178,1,1)	…	(-88,178,16,20)	<nodata>	…	<nodata>

```
$SE_HOME/bin/simenv_put_as bios_g.ascii
```

Since there is no second argument to simenv_put_as all variables without coordinate assignment (zero-dimensional variables) are output. This is only bios_g. The file has to have only one record with data and it must hold one data value.

The example model world_as.f writes the model output files atmo_bios.ascii, atmo_g.ascii and bios_g.ascii in the structures as explained above.

Example 5.6 *ASCII file structure for the ASCII model interface*

An example can be found in Section 15.2.12.

5.10 Semi-Automated Model Interface

Source code manipulations of a model for interfacing it to SimEnv can be classified into four parts:

- Initialization: apply simenv_ini_* and simenv_get_run_*
- Factor adjustments: apply simenv_get_*
- Model output: apply simenv_slice_* and simenv_put_*
- End: apply simenv_end_*

Often, “Initialization” and “Factor adjustments” can be lumped together in a source code sequence where the factor adjustment part has to be updated when new factors are defined in an experiment description file and have to be mapped to model internal factors the first time. Contrarily, “Model output” and “End” are often distributed in the model source code but do not change so often.

Recognising this situation SimEnv offers beside the standard hand-coded model interface a semi-automated model interface: “Initialization” and “Factor adjustments” are generated automatically during experiment preparation as sequences of source code based on the current experiment description file (and consequently the current experiment factors) for the Fortran, C/C++, Python, shell script and ASCII file model interface. For GAMS and Mathematica SimEnv offers such a simple model interface that a semi-automated interface is needless. For Java and Matlab there is no semi-automated SimEnv model interface as these two languages do not support include files.

These source code sequences can be used

- for Fortran/C/C++/Python model source codes
  as include files in the model source code and/or
- for the model interface at the shell script level and ASCII level
  as a dot script in <model>.run

to interface the model and consequently to run the experiment with an up-to-date part for initialization and factor adjustment.

For

- Fortran/C/C++ models:
  The model has to be compiled and linked anew with a new include file. This is supported by SimEnv in the course of experiment preparation.
- Python models and the model interface at shell script level and ASCII level:
  The include file and/or dot script can be used directly.

Generating source code sequences for the semi-automated model interface is invoked by the sub-keyword ‘auto_interface’ of the keyword ‘model’ in the model configuration file <model>.cfg (cf. Section 10.1).

The Fortan/C/C++/Python model interfaces offer to use different names of corresponding factors in the model description file <model>.edf and in the model source code that are associated by the first argument of the interface function simenv_put_* (cf. Fig. 5.1). **When using the semi-automated model interface the SimEnv factor names and the corresponding source code variable names have to coincide.**

Automatically generated source code sequences are stored in files <model>_[f | c | sh | as].inc and <model>_py.py in the current workspace $SE_WS. Note the file name exception for Python.

When using k factors x_1 … x_k in the experiment description file <model>.edf the source code sequences have the following contents:

for Fortran:
file <model>_f.inc

```
simenv_sts = simenv_ini_f       ( )
simenv_sts = simenv_get_run_f   ( simenv_run_int , simenv_run_char )
simenv_sts = simenv_get_f       ( ‘x1’ , 0. , x1 )
…
simenv_sts = simenv_get_f       ( ‘xk’ , 0. , xk )
```

for C/C++:
file <model>_c.inc

```
simenv_sts = simenv_ini_c         ( )
simenv_sts = simenv_get_run_c  ( &simenv_run_int , simenv_run_char )
simenv_sts = simenv_get_c         ( “x1” , &simenv_zero , &x1 )
…
simenv_sts = simenv_get_c         ( “xk” , &simenv_zero , &xk )
```

for Python:
file <model>_py.py

```
from simenv import *
simenv_ini_py ( )
simenv_run_int = int ( simenv_get_run_py ( ) )
x1 = float ( simenv_get_py ( ‘x1’ , 0. ) )
…
xk = float ( simenv_get_py ( ‘xk’ , 0. ) )
```

for the model interface at shell script level:
file <model>_sh.inc

```
. $SE_HOME/bin/simenv_ini_sh
. $SE_HOME/bin/simenv_get_run_sh
factor_name='x1'
factor_def_val=0.
. $SE_HOME/bin/simenv_get_sh
…
factor_name='xk'
factor_def_val=0.
. $SE_HOME/bin/simenv_get_sh
```

for the model interface at ASCII level:
file <model>_as.inc

```
. $SE_HOME/bin/simenv_ini_sh
. $SE_HOME/bin/simenv_get_run_sh
. $SE_HOME/bin/simenv_get_as
```

The sequence of factors in the code sequences corresponds to the sequence of factors in the experiment description file <model>.edf.

For the Fortran/C/C++ model interface

- a model link file <model>.lnk can be declared in the current workspace to link the model anew using the generated code sequences in the course of experiment preparation (only for service simenv.run, not for service simenv.rst).
- the variables simenv_sts, simenv_run_int, simenv_run_char, and simenv_zero are defined in the model source code include file simenv_mod_auto_[f | c].inc (cf. Tab. 5.10). Additionally, the functions simenv_[ini | get | get_run | put | slice | end]_[f | c] are declared by simenv_ mod_auto_[f | c].inc as integer*4 / int functions.

Tab. 5.10 *Built-in variables by simenv_mod_auto_[f | c].inc (without declaration of interface functions)*

Variable	Data type	Used for
simenv_sts	integer*4 / int	SimEnv interface function value
simenv_run_int	integer*4 /int	single run number
simenv_run_char	character*6 / char[6]	6 digit single run number string
simenv_zero	real*4 / float	auxiliary variable, set to 0.

The source code sequences are included in the model source code

for Fortran	by	include '<model>_f.inc'
for C/C++	by	#include "<model>_c.inc"
for Python	by	from <model>_py import *
for the model interface at shell script level	by	. $SE_WS/<model>_sh.inc
for the model interface at ASCII level	by	. $SE_WS/<model>_as.inc

Examples can be found in Example 15.2 and Example 15.12.

5.11 Supported Model Structures

SimEnv supports performance of lumped, distributed and parallel models. Information about model structure is specified in the model configuration file <model>.cfg (cf. Section 10.1) by the sub-keyword 'structure'. Lumped (standard) models are normally represented by one stand-alone executable. A distributed model in SimEnv consists from a web of stand-alone sub-models, i.e., the model dynamics are computed by performing a set of stand-alone sub-models that normally interact with each other and exchange information. For a parallel model each single run of an experiment needs a set of assign processors.

Lumped (standard) models use in the common sense SimEnv model interface functionality.

For distributed models each of the sub-models can use SimEnv model interface functionality, i.e., simenv_get_*, simenv_get_run_*, simenv_put_*, or simenv_slice_*. In each sub-model with SimEnv model interface functionality simenv_ini_* and simenv_end_* calls have to be incorporated in. Sub-models can be implemented in different programming languages. Additionally, the corresponding SimEnv model interface functionality at shell script level (simenv_*_sh dot scripts) can be applied. As usual, the overall model is wrapped into a shell script <model>.run (cf. Chapter 7).
The model output description file <model>.mdf collects all the model output variables from all sub-models and the experiment description file <model>.edf collects all the factors from all sub-models.
Announce a distributed model to the simulation environment if

- More than one sub-model uses SimEnv model interface functionality by the simenv_*_*-functions and
- Sub-models get factor data from and put model output data to SimEnv data files in parallel. A distributed model where the sub-models are performed sequentially one by one in a cascade-like manner can run in standard mode.

SimEnv interfaced sub-models of a distributed model can reside on different machines. The only prerequisite is that the current workspace and the model output directory can be mapped to each of these machines.

To perform a parallel model within SimEnv simply use the same approach for wrapping a model by the shell script file <model>.run as for standard and distributed models. Instead performing the model within <model>.run submit it there to the load leveler LoadL by using the llsubmit command. Start an experiment from a login-node of the compute cluster and run the experiment at the login machine in foreground. SimEnv submits from the login machine all single runs to LoadL and directly finishes afterwards. The load leveler LoadL and the parallel operating environment POE then take responsibility for performing the single model runs.
For the parallel modus the temporary SimEnv files simenv_*.tmp are not deleted at experiment end, i.e. after all single model runs are submitted. These files can be removed manually after finishing the last single run. Check the LoadL services for the end of the last parallel single model run.
To support bookkeeping of SimEnv applications on PIK's parallel cluster machine insert into the job control file to submit a single model run (file my_parallel_model.jcf in the example below) the line

```
# @ comment = SimEnv Application
```

Set the model sub-keyword 'structure' also to "parallel" if the model is to be started in the background (e.g., by my_model &) within <model>.run.

To perform a parallel model in SimEnv the corresponding shell script <model>.run (cf. Section 7.1 for more information) could have the following contents:

```
#! /bin/sh
. $SE_HOME/bin/simenv_ini_sh

# run a single run of the model:
llsubmit my_parallel_model.jcf

. $SE_HOME/bin/simenv_end_sh
```

Example 5.7 *Shell script <model>.run for a parallel model*

5.12 Using Interfaced Models outside SimEnv

To run a model interfaced to SimEnv outside the simulation environment in its native mode as before code adaptation the following simple changes have to be applied to the model:

- For Fortran and C/C++ models:
  Link the model with the object library
  $SE_HOME/lib/libsimenvdummy.a
  instead of
  $SE_HOME/lib/libsimenv.a.
  For this library
  - SimEnv model interface function values (return codes) are 0
  - simenv_get_* forwards factor_def_val to factor_adj_val
  - simenv_get_run_* returns integer run number 0 and character run string ‘ ‘ (six spaces).

- For Python models:
  Replace in the model source code
  from simenv import *
  by
  from simenvdummy import *
  For this modules
  - simenv_get_py forwards factor_def_val to factor_adj_val
  - simenv_get_run_py returns run 000000.
  - all other SimEnv model interface function values (return codes) are 0

- For Java models:
  Replace in the model source code
  the class Simenv
  by
  the class Simenvdummy
  From this class
  - simenv_get_py forwards factor_def_val to factor_adj_val
  - simenv_get_run_py returns run 000000.
  - all other SimEnv model interface function values (return codes) are 0

- For Matlab models:
  Replace in the model source code
  the Matlab function names simenv_[ini | get_run | get | slice | put | end]_m
  by
  simenvdummy_[ini | get_run | get | slice | put | end]_m
  From these functions
  - simenvdummy_get_py forwards factor_def_val to factor_adj_val

- simenvdummy_get_run_py returns run 000000.
- all other SimEnv model interface function values (return codes) are 0

- For Mathematica models:
  No changes required

- For GAMS models:
  Handle in the model source code the include statements
  - $include <model>_simenv_get.inc
  - $include <model>_simenv_put.inc

  as a comment.

6 Experiment Preparation

Experiment preparation is the first step in experiment performance of a model interfaced to the environment. In an experiment description file <model>.edf all information to the selected experiment type and its numerical equipment is gathered in a structured way.

6.1 General Approach - Experiment Description File <model>.edf

Pre-formed experiment types are the backbone of the SimEnv approach how to use models. They represent in a generic way experiment tasks that have to be equipped with structural in formation from the model and numerical information (cf. Chapter 4). An equipped experiment type is represented by an experiment description file <model>.edf.

<model>.edf is an ASCII file that follows the coding rules in Section 11.1 on page 141 with the keywords, names, sub-keywords, and value as in Tab. 6.1.

Tab. 6.1 *Elements of an experiment description file <model>.edf*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	experiment description
		type	m	1	[global sensitivity | behaviour | local sensitivity | monte carlo | optimization]	experiment type
factor	<factor_ name>	descr	o	1	<string>	factor description
		unit	o	1	<string>	factor unit
		type	m	1	see Tab. 6.2	factor adjustment type: specifies how to adjust the sampled values by the factor default value in simenv_get_* to get the resulting adjusted factor value
		default	m	1	<real_val>	factor default value <factor_def_val>
		sample	c3	1	<experiment specific>	specifies how to sample the factor to get sampled values <factor_smp_val>
specific	<nil>	<experiment specific>	m	<experiment specific>	<experiment specific>	experiment specific information

To Tab. 6.1 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- A factor name is the symbolic parameter / driver / initial value / boundary value name, corresponding to factors of the investigated model. Correspondence is achieved by applying the SimEnv model interface function simenv_get_* in the model.
- Factor names must differ from model output variables and coordinate names in the model output description file (cf. Section 5.1) and from built-in and user-defined operator names for experiment post-processing (cf. Section 8.5.4).
- **To derive the adjusted value of a factor its default value as specified in <model>.edf and not its default value from the model code is used in the model interface function simenv_get_*.**
- For the factor adjustment types 'multiply' and 'relative' a default <real_val> = 0. is forbidden.
- All experiment specific information is explained in the appropriate Sections.
- Specify at least one experiment factor.
- When preparing an experiment an experiment input file <model>.smp is generated with the sampled values <factor_smp_val> according to the information in the sub-keyword 'sample'. These values are applied within the interface function simenv_get_* to the default values of the factors according to the specified factor adjustment type (cf. Tab. 6.2 below) before finally influencing the dynamics of the model. The sequence of elements (columns) of each record of <model>.smp corresponds to the sequence of factors in the factor name space (cf. Section 11.1 on page 141), the sequence of records corresponds to the sequence of single model runs of the experiment.
- For each experiment a single model run with run number 0 (<simenv_run_int> = 0 and <simenv_run_char> = '000000') is generated automatically as the default (nominal) run of the model without any factor adjustments. This run does not have an assigned record in <model>.smp.

Tab. 6.2 *Factor adjustment types in experiment preparation*

Factor adjustment type	Meaning:
	To derive the final adjusted factor value <factor_adj_val> to use in the model from the sampled value <factor_smp_val> (from <model>.smp) and the factor default value <factor_def_val> (as defined in <model>.edf) within the SimEnv model interface function simenv_get_* the sampled value is …
set	… set to the adjusted factor value: **<factor_adj_val> = <factor_smp_val>**
add	… added to the factor default value: **<factor_adj_val> = <factor_smp_val> + <factor_def_val>**
multiply	… multiplied by the factor default value: **<factor_adj_val> = <factor_smp_val> * <factor_def_val>**
relative	… increased by 1 and afterwards multiplied by the factor default value: **<factor_adj_val> = (1. + <factor_smp_val>) * <factor_def_val>**

```
general          descr        Experiment description file
general          descr        examples
general          type         behaviour

factor     p1    descr        parameter p1
factor     p1    unit         without
factor     p1    type         set
factor     p1    default      1.
factor     p1    sample       (experiment specific)
```

```
factor     p2    type          set
factor     p2    default       2.
factor     p2    sample        1:10

specific         (experiment specific)
```

Example 6.1 *General layout of an experiment description file <model>.edf*

6.2 Global Sensitivity Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 53 is defined for local sensitivity analysis as follows:

Tab. 6.3 *Experiment specific elements of an edf-file for a global sensitivity experiment*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
factor	<factor_ name>	sample	m	1	<real_val$_1$>: <real_val$_2$>	lower bound <real_val$_1$> and upper bound <real_val$_2$> to define the sensitivity range where trajectories are positioned <real_val$_1$> ≤ <real_val$_2$> Factor values <factor_smp_val> are sampled within this sensitivity range.
specific	<nil>	levels	m	1	<int_val>	number of levels p ≥ 2 to define a p-level grid in the factor cube that is spanned up by the sensitivity ranges of the factors.
		trajectories	m	1	<int_val>	number of trajectories r ≥ 5 to position randomly at the p-level grid

To Tab. 6.3 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- To ensure that trajectories do not have to share grid points the ratio between the number of available grid points p^k in the factor cube and the number of points of the trajectories r*(k+1) has to be greater or equal than 3:
  p^k / r*(k+1) ≥ 3

6.2.1 Special Features in Global Sensitivity Analysis, Run Sequence

The sampling algorithm in preparing a global sensitivity experiment tries to avoid a multiple usage of grid points for the construction of the trajectories. If this fails after 20 trials a sample will be used that has at maximum five grid points in common in trajectories. A warning will inform about this situation.

The sequence of the single simulation runs in the experiment is determined in the following manner:

loop over trajectories

```
                loop            over over successive sampling points
                end loop
    end loop
```

6.2.2 Example

```
(2) general          descr         Experiment description for the examples
    general          descr         in the SimEnv User Guide
    general          type          global sensitivity

    factor     p1    descr         parameter p1
    factor     p1    unit          without
    factor     p1    type          set
    factor     p1    default       1.
    factor     p1    sample        -12:12    check sensitivity for factor p1 in <–12 , 12>

    factor     p2    type          set
    factor     p2    default       2.
    factor     p2    sample        1:10
    factor     p3    type          set
    factor     p3    default       3.
    factor     p3    sample        -12:12
    factor     p4    type          set
    factor     p4    default       4.
    factor     p4    sample        1:10

    specific         levels        4
    specific         trajectories  10
```

Example file: world.edf_2

Example 6.2 *Experiment description file <model>.edf for a global sensitivity analysis*

6.3 Behavioural Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 53 is defined for behavioural analysis as follows:

Tab. 6.4 *Experiment specific elements of an edf-file for behavioural analysis*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
factor	<factor_ name>	sample	a	1	<val_list>	value list of factor samples <factor_smp_val> (for syntax see Tab. 11.6)
specific	<nil>	comb	m	≥ 1	[default | <combination> | file {<directory>/} <file_name> { [strict | nonstrict] }]	information how to scan the spanned factor space

To Tab. 6.4 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- For sub-keyword 'comb' the following rule holds:
  value = [default | <combination>] for used sub-keyword 'sample'
  value = [file {<directory>/}<file_name>] for unused sub-keyword 'sample'
- Values of a value list have to be unique for used sub-keyword 'sample' and each factor
  Assigned values from file {<directory>/}<file_name> can be multiple defined for each factor.

The sequence of the single runs is determined by the sub-keyword 'comb'.

6.3.1 Formalisation of the Inspection Strategy, Run Sequence

- The combination **<combination>** defines the way in which the space spanned by the experiment factors will be inspected by SimEnv. This is done by concetenating all stated experiment factors by operators „*" and „,".
  - **The operator „*"** combines sampled values of different factors and so their resulting adjusted values combinatorially ("the Cartesian product of the sampled values of all factors").
    For example, compare with the experiment description file (3a) from Example 6.3 below.
  - **The operator „,"** combines sampled values of different factors and so their resulting adjusted values in parallel ("on the diagonal in the space spanned up from all factors").
    For the operator „," the factors must have the same number of sampled values.
    For example, compare with the experiment description file (3b) from Example 6.3 below.
  - The operators „," and „*" can be multiple used in <combination>. The operator „," has a higher priority than the operator „*". Parentheses are not allowed:
    For example, x_1 * x_2 , x_3 * x_4 always combines factors x_2 and x_3 in parallel and this combinatorially with factors x_1 and x_4. A parallel combination of x_1 * x_2 with x_3 * x_4 by (x_1 * x_2) , (x_3 * x_4) is not possible.
    For example, compare with the experiment description file (3c) from Example 6.3 below.
  - In <combination> each factor has to be used exactly once.
- By the default combination default all experiment factors are combined combinatorially in the sequence of their declaration in the experiment description file.
  - For example, comb default of the experiment description file (3a) from Example 6.3 below is equivalent to comb p1 * p2 .
- Specification of **file** in the comb is only allowed if sub-keywords 'sample' were not specified for all factors in the edf-file.
  - All factors are assumed to be combined in parallel.
  - The sampled values are read from the sample data file **{<directory>/}<file_name>**.
  - Each record of the sampled values data file represents one simulation run. The sequence of the sample (sequence of columns) in each record corresponds to the sequence of the factors in the factor name space (cf. Section 11.1 on page 141).
  - Consequently, the file has to have per record as much values as factors defined in <model>.edf. All the other syntax rules for ASCII data files from Section 11.3 hold.
  - When specifiying **{<directory>/}<file_name> strict** or **{<directory>/}<file_name>** identical sample values for a factor are not allowed to enable after the experiment SimEnv post-processing with the experiment-specific multi-run operator behav.
  - When specifiying **{<directory>/}<file_name> nonstrict** identical sample values for a factor are allowed. Experiment-specific multi-run operator behav can not be applied in post-processing. Use ens instead.
  - During experiment post-processing restricted capabilities for the operator behav apply for this experiment layout.
  - For example, compare with the experiment description file (3d) from Example 6.3 below. Combination is implicitly as comb p1 , p2. Experiment description files (3b) and (3d) in Example 6.3 below describe the same experiment.
- To continue a combination <combination> at a following comb-line end the current comb-line by one of the operators "*" or ",".
- An explicit stated combination <combination> is normalized before running the experiment in the following sense:
  - Segments of <combination> that are separated by the operator „*" can be re-arranged in an arbitrary order. For example, p2 * p1 is equivalent to p1 * p2.

- Factors that are scanned in parallel can be re-arranged in an arbitrary order. For example, p4 , p3 * p2 , p1 is equivalent to p3 * p4 * p2 , p1.
- <combination> is rearranged in a way that factors are used in the sequence they are declared in the experiment description file. For example, if four factors are declared in the dequence p1 , p2 ,p3 , p4 then the explicitly stated <combination> p4 , p2 * p3 , p1 is normalized to p1 , p3 * p2 , p4.
- Nomalization does not influence the layout of the experiment.

The sequence of the single simulation runs in the experiment is determined in the following manner:

- For comb file {<directory>/}<file_name> :
  The sequence corresponds to the sequence of the sampled factor values in the file <file_name>.
- For comb <combination>
  with the normalized <combination> = $<x_1>$ * $<x_2>$ * … * $<x_n>$ and
  $<x_i> = \{ x_{i1} , x_{i2} ,\ldots, x_{ii^*} \} := \{ x_{ij} \}_{j=1,\ldots,i^*}$ for I = 1 ,…, n

```
loop            over all factor sample values { x_nj }_j=1,...,i* for <x_n>
                …
                loop            over all factor sample values { x_2j }_j=1,...,2* for <x_2>
                                loop            over all factor sample values { x_1j }_j=1,...,1* for <x_1>
                                end loop
                end loop
                …
end loop
```

- For comb default :
  Is put down to comb <combination> (see above)

6.3.2 Example

Experiment description file (3a) represents an experiment description according to Fig. 4.4 (a) on page 18, (3b) and (3d) according to Fig. 4.4 (b) and (3c) according to Fig. 4.4 (c).

					Results in adjusted factor values
(3a)	general		descr	Experiment description for the examples	
	general		descr	in the SimEnv User Guide (Fig. 4.4 (a))	
	general		type	behaviour	
	factor	p1	descr	parameter p1	
	factor	p1	unit	without	
	factor	p1	type	add	
	factor	p1	default	1.	
	factor	p1	sample	list 1, 3, 7, 8	... 2, 4, 8, 9 for p1
	factor	p2	descr	parameter p2	
	factor	p2	unit	without	
	factor	p2	type	multiply	
	factor	p2	default	2.	
	factor	p2	sample	list 1, 2, 3, 4	... 2, 4, 6, 8 for p2
	specific		comb	default	
(3b)	general		descr	Fig. 4.4 (b)	
	general		type	behaviour	
	factor	p1	type	multiply	
	factor	p1	default	1.	
	factor	p1	sample	list 1, 3, 7, 8	... 1, 3, 7, 8 for p1

```
     factor     p2    type        multiply
     factor     p2    default     2.
     factor     p2    sample      equidist_end 1(0.5)2.5        ... 2, 3, 4, 5 for p2
     specific         comb        p1,p2

(3c) general          descr       Fig. 4.4 (c)
     general          type        behaviour
     factor     p1    type        set
     factor     p1    default     1.
     factor     p1    sample      list 1, 3, 7, 8               ... 1, 3, 7, 8 for p1
     factor     p2    type        set
     factor     p2    default     2.
     factor     p2    sample      equidist end 1(1)4            ... 1, 2, 3, 4 for p2
     factor     p3    type        multiply
     factor     p3    default     3.
     factor     p3    sample      list 2.0, 2.8, 3.3            ... 6.0, 8.4, 9.9 for p3
     specific         comb        p1,p2*p3

(3d) general          descr       Fig. 4.4 (b)
     general          type        behaviour          file world.dat 3d:
     factor     p1    type        multiply             1     0
     factor     p1    default     1.                   3     1
     factor     p2    type        add                  7     2
     factor     p2    default     2.                   8     3
     specific         comb        file world.dat_3d strict      ... (1,2), (3,3), (7,4),
                                                                    (8,5) for (p1,p2)

                                                  Example files: world.edf_3a to world.edf_3d
```

Example 6.3 *Experiment description file <model>.edf for behavioural analysis*

6.4 Local Sensitivity Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 53 is defined for local sensitivity analysis as follows:

Tab. 6.5 *Experiment specific elements of an edf-file for local sensitivity analysis*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
factor	<factor_ name>	sample	f	0		sub-keyword is forbidden for this experiment type
specific	<nil>	incrs	m	1	<val_list>	Increments that form a sample of factor values <factor_smp_val>. Resulting <factor_smp_val> from <val_list> have to be ordered in a strictly monotonic increasing manner. (for syntax see Tab. 11.6)

To Tab. 6.5 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- For experiment type local sensitivity analysis only the factor adjustment types ‘add’ and ‘relative’ are allowed.
- Values from the value list must be positive and unique.

6.4.1 Sensitivity Functions, Run Sequence

As an example, the absolute sensitivity function (cf. Tab. 4.2 on page 19) is as follows:

for adjustment type Add

$$\text{sens\_abs}(\langle\text{factor\_def\_val}\rangle,\pm\langle\text{factor\_smp\_val}\rangle) = \frac{z(\langle\text{factor\_def\_val}\rangle \pm \langle\text{factor\_smp\_val}\rangle) - z(\langle\text{factor\_def\_val}\rangle)}{\pm \langle\text{factor\_smp\_val}\rangle}$$

for adjustment type Relative

$$\text{sens\_abs}(\langle\text{factor\_def\_val}\rangle,\pm\langle\text{factor\_smp\_val}\rangle) = \frac{z(\langle\text{factor\_def\_val}\rangle * (1\pm \langle\text{factor\_smp\_val}\rangle) - z(\langle\text{factor\_def\_val}\rangle)}{\pm \langle\text{factor\_def\_val}\rangle * \langle\text{factor\_smp\_val}\rangle}$$

The sequence of the single simulation runs in the experiment is determined in the following manner:

```
loop            over increment sequence
                loop            over experiment factors
                end loop
end loop
loop            over negative increment sequence
                loop            over experiment factors
                end loop
end loop
```

6.4.2 Example

```
(4) general          descr      Experiment description for the examples
    general          descr      in the SimEnv User Guide
    general          type       local sensitivity

    factor     p1    descr      parameter p1
    factor     p1    unit       without
    factor     p1    type       add
    factor     p1    default    1.

    factor     p2    type       relative
    factor     p2    default    2.
    factor     p3    type       relative
    factor     p3    default    3.

    specific         incrs      list 0.001,0.01,0.05,0.1
```

Example file: world.edf_4

Example 6.4 *Experiment description file <model>.edf for local sensitivity analysis*

6.5 Monte Carlo Analysis

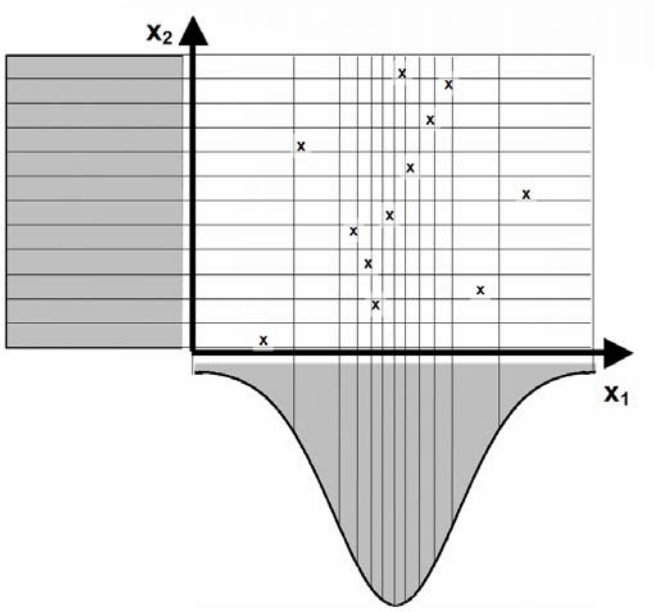
The experiment specific information for experiment description files in Tab. 6.1 on page 53 is defined for Monte Carlo analysis as describes in Tab. 6.6.

Tab. 6.6 *Experiment specific elements of an edf-file for Monte Carlo analysis*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
factor	<factor_ name>	sample	m	1	[<distribution> | file {<directory>/} <file_name>]	distribution and distribution parameters to derive a sample of values <factor_smp_val> or file name to import an external sample of values <factor_smp_val>
		sample_ method	c4	1	[random | latin hypercube]	sampling strategy: random or latin hypercube sampling LHS
specific	<nil>	runs	m	1	<int_val>	number of runs > 10 to be performed for the experiment
		function	o	≥ 0	<result>	stopping function to use in a stopping rule for the experiment. A 0-dimensional result formed according to the rules of the SimEnv post-processor. Do not apply multi-run operators. Stopping function definition can be arranged at a series of function-lines in analogy to the rules for result expressions (cf. Section 8.1.1).

To Tab. 6.6 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- For <distribution> = <distr_shortcut> (<distr_param_1> { , <distr_param_2> }) check Tab. 6.7.
- For implicitly specified distributions according to Tab. 6.7 sample values <factor_smp_val> are generated from the distribution with the assigned distribution parameters.
- If an ASCII file {<directory>/}<file_name> is stated the sample values of any distribution are taken directly from this file. Each record of the ASCII file can hold only one sample value. For the other syntax rules for ASCII data files check Section 11.3. Sample size has to be identical to <int_val> from the sub-keyword 'runs'.
- In random sampling, there is no assurance that sampling points will cover all regions of the selected distribution. With Latin hypercube sampling LHS (McKay *et al.*, 1979) this shortcoming is reduced: The sampling range of the factor is divided into <int_val> (from the sub-keyword 'runs') intervals of equal probability according to the selected distribution and from each interval exactly one sampling point is drawn. For more information on LHS check Fig. 6.1 below and see also Iman & Helton (1998) and Helton & Davis (2000).
- The number of runs must be greater than 10.



Latin hypercube sampling
for a sample size of
12 single simulation runs.

Factor x_1 is normally distributed
Factor x_2 is uniformly distributed.

Compare with random sampling in
Fig. 4.6 on page 20.

Fig. 6.1 *Monte Carlo analysis: Latin hypercube sampling*

6.5.1 Distribution Functions and their Parameters, Stopping Rule

Tab. 6.7 *Probability density functions and their parameters*

Distribution function	distr_ shortcut	distr_param_1	distr_param_2	Restriction
uniform	U	lower boundary	upper boundary	lower boundary < upper boundary
normal	N	mean value	variance	variance > 0
lognormal	L	mean value of a normally distributed factor	variance of a normally distributed factor	variance > 0
exponential	E	mean value	---	mean value > 0

For more information on the distribution functions see Section 4.4 and Tab. 4.4.

Be careful when specifying for a Monte Carlo analysis a factor adjustment type (cf. Tab. 6.2) that differs from ‘set’. Values are sampled according to the specified distribution and its declared distribution parameters and/or are used from the input files. Nevertheless, each value of the sample is modified according to the factor adjustment type in simenv_get_*. So, for the factor adjustment type ‘add’ normally the mean value of the sample will be shiftet by the specified factor default value <factor_def_val>. For the factor adjustment types ‘multiply’ and ‘relative’ the specified distribution will be adulterated normally by the factor default value <factor_def_val>.

Optionally, SimEnv enables definition of a stopping rule that can be helpful to limit the number of simulation runs in an experiment. In a stopping rule statistical measures from all already performed single model runs of

the run ensemble are calculated after each single run to decide whether to stop the whole experiment. Statistical measures are computed from a 0-dimensional result res(z) (the stopping function) formed according to the rules of the SimEnv post-processor. The stopping function is used as an indicator to stop the experiment.

In SimEnv the point of change for the variance of the stopping function res(z) over the already perfomed single runs is determined after each single run using the Pettitt test (Pettitt, 1979). If a point of change in the sequence of the single runs over the already performed run ensemble is detected, it is assumed that the variance of the stopping function does not change anymore significantly after the point of change. The first half of the simulation runs of the experiment is perfomed without applying the test in order to generate a stabilized stopping function sample res(z).

The whole experiment is stopped if

- the level significance of the Pettitt test is below 0.05 for the already performed run ensemble and
- there were at least <int_val>/5 single runs after that single run that represents the point of change. <int_val> is the number of declarad runs in <model>.edf (see above).

The latter condition is introduced to avoid to run into a local point of change.

Monte Carlo experiments with a stopping function cannot be re-started. Partial experiment performance is not supported. Consequently, in the configuration file <model>.cfg sub-keywords 'begin_run' / 'end_run' / 'include_runs' / 'exclude_runs' are not allowed for an experiment with a stopping function. The stopping condition is reported to the experiment log-file <model>.elog.

6.5.2 Example

```
(5) general        descr      Experiment description for the examples
    general        descr      in the SimEnv User Guide
    general        type       Monte Carlo

    factor    p2   descr      parameter p1
    factor    p2   unit       without
    factor    p2   type       multiply
    factor    p2   default    2.
    factor    p2   sample_method    latin hypercube
    factor    p2   sample     distr U(0.5,1.5)    p2 is sampled from a uniform distrib.
                                                   between 0.5 and 1.5. In simenv_get_*
                                                   each value is multiplied by 2.

    factor    p1   type       add
    factor    p1   default    1.
    factor    p1   sample_method    random
    factor    p1   sample     distr N(0,0.4)      p1 is sampled from a normal distribution
                                                   with mean = 0. and variance = 0.4.
                                                   In simenv_get_* each value is increased
                                                   by 1.

    factor    p3   type       set
    factor    p3   default    3.
    factor    p3   sample     file world.dat_5    sample for p3 is read from file
                                                   world.dat_5

    specific       runs       250
    specific       function   avg(atmo_g)         avg(atmo_g) as stopping function
```

Example file: world.edf_5

Example 6.5 *Experiment description file <model>.edf for Monte Carlo analysis*

6.6 Optimization

The experiment specific information for experiment description files in Tab. 6.1 on page 53 is defined for local sensitivity analysis as follows:

Tab. 6.8 *Experiment specific elements of an edf-file for an optimization experiment*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
factor	<factor_ name>	sample	m	1	<real_val$_1$>: <real_val$_2$>	lower bound <real_val$_1$> and upper bound <real_val$_2$> to define the factor range where the cost function is to be minimized on. <real_val$_1$> ≤ <real_val$_2$> Values <factor_smp_val> are sampled in this factor range.
specific	<nil>	function	m	≥ 1	<result>	cost function to minimize. A 0-dimensional result formed according to the rules of the SimEnv post-processor. Do not apply multi-run operators. Cost function definition can be arranged at a series of function-lines in analogy to the rules for result expressions (cf. Section 8.1.1).
		runs	m	1	<int_val>	number of single runs to end the experiment without checking the other optimization method related stopping criteria.

To Tab. 6.8 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.

6.6.1 Special Features in Optimization

- This is the only experiment type where the sample for the factors of the single runs are not determined before the experiment but in the course of the experiment by the optimization algorithm. Consequently, only the header of the file <model>.smp is created during experiment preparation. The records belonging to the performed single runs are written during experiment performance.
- In parallel to the file <model>.smp an ASCII file **<model>.fct** is written during experiment performance with the value of the cost function for each of the single runs.
- The optimization algorithm itself is controlled by additional technical parameters and options that are normally fixed by SimEnv. To modify these settings copy the ASCII file **simenv_opt_options.txt** from the sub-directory bin of the SimEnv home directory to <model>_opt_options.txt in the current workspace and edit this file. During the experiment the edited file is used instead of the file with the default constellation in the SimEnv home directory. The description of the options and parameters can be found in Ingber (2004).
- Optimization experiments cannot be restarted by the SimEnv service simenv.rst.
- In the configuration file <model>.cfg sub-keywords 'begin_run' / 'end_run' / 'include_runs' / 'exclude_runs' are not allowed for an optimization experiment. The experiment always starts with run num-

ber 0 and ends if one of the criteria in the file [<model> | simenv]_opt_options.txt (see above) is fulfilled or the explicitly stated end run number from the sub-keyword 'runs' in <model>.edf is reached.

- As the results of the optimization experiment the optimization return code, the optimal factors, the corresponding value of the cost function and the number of the corresponding single run are documented at the end of the model interface log-file <model>.mlog.
- A protocol from the optimization procedure is made available by SimEnv in the ASCII file <model>.olog.

6.6.2 Example

```
(6) general          descr     Experiment description for the examples
    general          descr     in the SimEnv User Guide
    general          type      optimization

    factor    p1     descr     parameter p1
    factor    p1     unit      without
    factor    p1     type      set
    factor    p1     default   1.
    factor    p1     sample    -12:12      minimize cost function for p1ϵ <–12 , 12>

    factor    p2     type      set
    factor    p2     default   2.
    factor    p2     sample    1:10
    factor    p3     type      set
    factor    p3     default   3.
    factor    p3     sample    -12:12
    factor    p4     type      set
    factor    p4     default   4.
    factor    p4     sample    1:10

    specific         function  -sum(bios)  maximize sum(bios) over land masses
    specific         runs      700
```

Example file: world.edf_6

Example 6.6 *Experiment description file <model>.edf for an optimization experiment*

7 Experiment Performance

After experiment preparation experiment performance is the second step in running a model interfaced to SimEnv. Each multi-run experiment can be performed sequentially or in a multi-processor hardware environment. Besides experiment performance from scratch a restart after an experiment interrupt or only for an experiment slice can be handled by SimEnv.

7.1 General Approach

SimEnv enables performance of an experiment in different modes: on the login-machine in foreground or in background and in a job class controlled by the load leveler LoadL and the parallel operating environment POE. Experiment performance on the login-machine is organized in a way that the single runs of the experiment are performed sequentially. Experiment control by LoadL and POE enables assignment of the simulation load of the single runs of the experiment to a number of processors in distributed, parallel or sequential mode.

Experiments may be performed partially only for a slice out of the run ensemble. Experiment slices are controlled by the general configuration file <model>.cfg by a range of single run numbers.

Experiments can be re-startet for successive performance of experiment slices and/or after abnomal experiment interrupt. The experiment log-file <model>.elog is analyzed to identify these single runs out of the run ensemble that have to be performed the first time and/or anew and the corresponding model output data is appended to the output data that already exists for this experiment.

For all experiment settings the user model has to be wrapped in a shell script <model>.run (cf. also Fig. 5.1).

Moreover, the following conditions are valid when running an experiment. For more details check the corresponding Sections.

- The model variables to be output during experiment performance are declared in the model output description file <model>.mdf
- The type and the factors of the experiment to be performed are declared in the experiment description file <model>.edf
- Mapping between experiment factors and factors in the model source code is achieved by application of the generic SimEnv model interface function simenv_get_* in the model code or at shell script level.
- Output of model variables declared in <model>.mdf into SimEnv structures is achieved by the application of the generic SimEnv model interface function simenv_put_* (and simenv_slice_*) in the model source code.
- Model output from run number <simenv_run_int> is stored in the file <model>.out<simenv_run_char>.[nc | ieee] if the value if the out_separation sub-keyword in <model>.cfg is set to ‘yes’. Otherwise, model output from the complete experiment is stored in <model>.outall.[nc | ieee].
- For all experiment types a run number 0 with the default values of all experiment factors will is declared additionally to the runs declared in the experiment description file <model>.edf.
- During experiment performance a model interface log-file <model>.mlog is written where the adjusted experiment factor values are logged. All model output to the terminal is re-directed within SimEnv to the experiment model native output log-file <model>.nlog.
- During experiment performance an experiment log-file <model>.elog is written with the minutes of the experiment.
- After the experiment has been finished an email is sent on demand (cf. Section 10.1) to the address as specified in <model>.cfg.
- The status of any running experiment can be acquired by the SimEnv service simenv.sts. For more information check Tab. 10.4.
- Do not start / restart / submit another experiment from a workspace where an experiment is still running.
- For more information check Section 5.1, Fig. 5.1 and Fig. 7.1.

7.2 Model Wrap Shell Script <model>.run, Experiment-Specific Preparation and Wrap-Up Shell Scripts

- The model to be applied within the SimEnv experiment has to be wrapped in the shell script **<model>.run**. <model>.run is performed for each single run within the run ensemble.
  - **Make sure that in <model>.run**
    - **#! /bin/sh is the first line**
    - **. $SE_HOME/bin/simenv_ini_sh is performed always and as the first SimEnv dot script**
    - **. $SE_HOME/bin/simenv_end_sh is performed always and as the last SimEnv dot script**

    (cf. Tab. 5.8 on page 44 and Example 7.1 below).
  - Terminal output from <model>.run is redirected to the log-file <model>.nlog.
  - For GAMS models <model>.run has a pre-defined structure. Check Section 5.7.1 for more information.
  - To cancel the whole experiment after the performance of the current single run <simenv_run_int> due to any condition of this run make sure a file **$SE_WS/<model>.err<simenv_run_char>** exists as an indicator to stop. Create this file in the model or in <model>.run. For the latter
    - Perform . $SE_HOME/bin/simenv_get_run_sh to get the current run number <simenv_run_int> and <simenv_run_char>
    - Touch the file $SE_WS/<model>.err<simenv_run_char>
    - Cf. Tab. 5.8 on page 44 and Example 7.1 below

    From the cancelled experiment only those single runs are available for experiment postprocessing that were finished before the cancelled single run. Check <model>.elog to identify these single runs.
  - SimEnv supplies a shell script simenv_kill_process to kill models / programs that were started within <model>.run and that consumed more than a given threshold of CPU time. For example, with this script models can be killed that do not converge and would run infinitely.
    Start this script in background directly before the process is started that is to be monitored:
    $SE_HOME/bin/simenv_kill_process <program_to_monitor> <CPU_time_threshold_in sec>
    When the program is killed a file **$SE_WS/<model>.killed<simenv_run_char>** exists as an indicator. Keep in mind that for killed models normally the status of model output to SimEnv data structures may be undefined. Sub-processes of the killed model are not killed by the shell script simenv_kill_process. Check Example 7.4.

- The user can define an optional model specific experiment preparation shell script **<model>.ini** that is performed additionally after standard experiment preparation and before setting up a new experiment. For experiment restart <model>.ini is performed only on request (cf. Section 7.4 below).
  - In <model>.ini additional settings / checks can be performed. For return codes unless 0 from <model>.ini the experiment will not be started.
  - Terminal output from <model>.ini is re-directed to the log-file <model>.nlog.
  - For Python, Java, Matlab and GAMS models <model>.ini is a mandatory shell script with standardized contents. Check Sections 5.5.1 and 5.7.1 for more information.

- After the experiment has been finished the native model specific output from the experiment can be wrapped up with the optional model specific shell script **<model>.end**.
  - Terminal output from <model>.end is re-directed to the log-file <model>.nlog.
  - For GAMS models <model>.end is a mandatory shell script with standardized contents. Check Section 5.7.1 for more information.

- All of these three shell scripts have to have execute permission. Ensure this by the Unix / Linux command
  chmod u+x <model>.[run | ini | end]

For the shell script world_f.run the following contents could be defined:

```
#! /bin/sh

# perform always and as the first $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_ini_sh

# run the model:
./world_f

# assuming a model return code ≠ 0 as an indicator to stop
# the whole experiment for any reason.
# Touch the file below in the current workspace $SE_WS
# as an indicator to SimEnv for this.
if test $? -ne 0
then
      . $SE_HOME/bin/simenv_get_run_sh
      touch $SE_WS/world_f.err$simenv_run_char
fi

# perform always and as the last $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_end_sh
```

Example file: world_f.run

Example 7.1 *Shell script <model>.run to wrap the user model*

For the shell script world_*.ini the following contents could be defined
(for Matlab, the coarsed land sea mask is restructured additionally):

```
# coarse 0.5° x 0.5° land-sea mask from file land_sea_mask.05x05
# in the current directory
# to a 4° x 4° resoluted land-sea-mask in file land_sea_mask.coarsed
# in the current directory to use for all single runs
./land_sea_mask 4 4
rc_land_sea_mask=$?

# exit from world_*.ini with return code ≠ 0
# as an indicator not to start the experiment
exit $rc_land_sea_mask
```

Example files: world_[f | c | cpp | py | ja | m | sh | as].ini

Example 7.2 *Shell script <model>.ini for user-model specific experiment preparation*

For the shell script world_f.end the following contents could be defined:

```
# remove the file of the coarsed land-sea mask
rm -f land_sea_mask.coarsed
```

Example file: world_[f | c | cpp | py | ja | m | sh | as].end

Example 7.3 *Shell script <model>.end for user-model specific experiment wrap-up*

For the shell script world_f.run the following contents could be defined:

```
#! /bin/sh

# perform always and as the first $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_ini_sh

# enable to kill the process associated with the model $SE_WS/world_f
# after 100 seconds of CPU time consumption
$SE_HOME/bin/simenv_kill_process $SE_WS/world_f 100 &

# run the model:
$SE_WS/world_f

# take some actions when the model was killed
. $SE_HOME/bin/simenv_get_run_sh
if test $SE_WS/world_f.killed$simenv_run_char
then
        . . .
        rm -f $SE_WS/world_f.killed$simenv_run_char
fi

# perform always and as the last $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_end_sh
```

Example file: world_f.run

Example 7.4 *Shell script <model>.run with shell script simenv_kill_process*

7.3 Experiment Performance on the Login Machine and under Job Management System Control

According to the general SimEnv approach how to design an experiment the single runs of an experiment are independent from each other. The only exception is the experiment type optimization where the sample values for the current single run are determined on the outcomes of previous single runs. Keeping this in mind, SimEnv offers different modes how to disseminate the single runs of an experiment during its performance. Experiments can run

- locally on the login machine
  in foreground and background mode and distributed on a multicore processor machine
- under control of the load leveler LoadL job management system
  in parallel, distributed and sequential mode

For an experiment performance controlled by the the job management system or on the login machine in background **make sure that the environment variable SE_HOME is set correctly in the file $HOME/.profile**.

Local experiment performance on the login machine
Two different distribution stategies are offered by the simulation environment for running an experiment on the login machine:
Perform the single runs of an experiment …

- … sequentially on the login machine
  foreg - foreground sub-mode
  backg - background sub-mode:
  For an experiment in foreground sub-mode the login session must be active during the whole experiment.

Choosing experiment performance in the background, a temporary shell script is generated by SimEnv that represents the simulation experiment as a whole. This shell script is submitted as a cron job to the cron daemon for one-time execution starting at a time specified during experiment preparation. The cron job is removed from the cron job list directly after the start of the corresponding experiment. After experiment preparation the login session can be closed. For background performance make sure to have rights to manage cron jobs on the machine the experiment is started from.

- … distributed on a multicore processor machine
  mcore - multi-core sub-mode:
  For a multi-core processor login machine the single runs of the experiment can be distributed across a selected number of cores. The experiment can be started at once or at a specified time. For the latter, a cron job is generated which is removed from the cron job list directly after the start of the corresponding experiment. For cron job submission make sure to have rights to manage cron jobs on the machine the experiment is started from. As for background sub-mode, in multi-core sub-mode the login session can be closed after experiment preparation.

Experiment performance controlled by the job management system
SimEnv enables the parallelization of the experiment in the sense that several single runs can be performed in parallel without influencing each other. This opens an approach for a computer network or a compute cluster of connected machines

- to distribute the single runs of an experiment across the network / on the cluster
- to perform the single runs there and
- to collect after the end of a single model run its model output data and related information

SimEnv supports distribution of single runs from an experiment for compute cluster architectures. Currently, IBM's Job management system load leveler LoadL and the parallel operating environment POE are supported by SimEnv. The processors of a compute cluster are assigned to job classes where jobs can be submitted to.

Two different distribution stategies are offered by the simulation environment:
Perform the single runs of an experiment …

- … on all the available processor cores of a job class
  dis - distributed sub-mode:
  The single runs are submitted to the job class as single jobs in a way that all available processor cores of the class can be used. Due to controlling the submit process dynamically by SimEnv, the job class will not be overloaded by the single run jobs of the experiment. Instead, the submit process will wait if necessary. The submit process itself is started in the background. The experiment performance will start with the first submitted single run when a processor core of the selected job class is free.
  Use this sub-mode for best utilization of all job class processor cores.

- … on pre-allocated processor cores of a class
  par - parallel sub-mode:
  A number of processor cores are assigned to the experiment during experiment preparation and one parallel job is submitted to the job class. During the experiment one communication processor node is responsible for experiment management while the other processors serve as simulation processor cores for the single runs.
  The experiment performance will start when the assigned number of processor cores are free in this class. This sub-mode makes use of the Message Passing Interface MPI.
  Use this sub-mode to make sure to run an experiment in a certain time.
  For inter-node communication, check the remark below.

- … on one pre-allocated processor core of a class
  seq - sequential sub-mode:
  Addionally to the above distributed and parallel sub-modes SimEnv also offers a sequential sub-mode under control of the job management system: One processor core of a job class is assigned to the whole experiment and the experiment is performed sequentially on this processor core. The experiment per-

formance will start when one processor core of this job class is free. After an experiment is submitted to the load leveler the current login session can be closed.

Default job control files are supplied by SimEnv to ensure communication with the load leveler and POE. These job control files may be copied to the current workspace, can be modified and will then be used instead of the default job control files to start an experiment with one of those parallelization strategies that use the load leveler and POE.
If necessary, copy the ASCII job control files **simenv_[dis | par | seq]_[aix | linux].jcf** from the sub-directory bin of the SimEnv home directory to <model>_[dis | par | seq]_[aix | linux].jcf in the current workspace, modify the file(s) according to the needs of the model one wants to perform and / or the machine one wants to use and start afterwards simenv.run and/or simenv.rst anew. If available in the current workspace, these modified job control files are used instead of the original files in the sub-directory bin of the SimEnv home directory. simenv_[dis | par | seq]_[aix | linux].jcf and/or <model>_[dis | par | seq]_[aix | linux].jcf submit a job in distributed / parallel / sequential sub-mode under LoadL control.

The default job control files and SimEnv log-files enable automatic restart of the experiment by the load leveler after an interrupt of the job caused by the operating system, the load leveler or POE. The user does not need to restart the experiment manually after such an event.

For performing a parallel model itself see Section 5.11.

Peculiarities of multi run experiment performance
Contrary to a single model run, a native model source code has to be analysed at least with respect to its output files before setting up a multi-run simulation experiment. Often, models write output to files with fixed file names and these files must not exist before running the model. Such assumptions conflict with running the model in a loop sequentially or in parallel / distributed sub-mode.

Pragmatic workarounds for such conditions without changing the model source code are as follows:
- For sequential experiment performance on the login machine and/or on a compute cluster
  rename in the model wrap shell script <model>.run after running the model its outputs to run number related file names. This solves most of the problems since always only one model run is active.
- For parallel and distributed experiments on the login machine and/or on a compute cluster
  this solution failes since more than one model run is active and output files are opened. Here , the best choice is to perform each single model run in its own (temporary) subdirectory of the current workspace, e.g. identified by the number of the single run. Keep in mind that input files also have to be copied to this directory.

Check Example 7.5 for more information.

Inter-node communication for parallel sub-mode at compute clusters:
The Message Passing Interface MPI is used for this sub-mode. To start the simenv binary $SE_HOME/bin/simenv_run_par, MPI needs ssh-connections between the nodes / blades of the cluster. The ssh-connections need public and private keys and appropriate authorization entries.
At the PIK compute clusters openssh is used. openssh uses the directory ~/.ssh for key files. A minimal directory contents of ~/.ssh looks like this:

```
login02a:~> ls
id_[ d | r ]sa                  private key
id_[ d | r ]sa.pub              public key
authorized_keys                 file of accepted public keys
```

id_[d | r]sa.pub must be authorized_keys.
Pay attention that id_[d | r]sa and id_[d | r]sa.pub are really a key pair.
It is recommended to keep the directories ~/.ssh and ~/.ssh2 disjunct.

For a model my_model with an input file my_model.in and an output file my_model.out the following contents could be defined for the model wrap shell script my_model.run:

```
#! /bin/sh

# perform always and as the first $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_ini_sh

# get run number
. $SE_HOME/bin/simenv_get_run_sh

# for sequential experiment performance:
./my_model
mv my_model.out my_model.out.$simenv_run_char
# sequential end

# for parallel and distributed experiment performance:
mkdir run$simenv_run_char
cd run$simenv_run_char
cp ../my_model.in .
../my_model
mv my_model.out ../my_model.out.$simenv_run_char


cd ..
rmdir run$simenv_run_char
# parallel and distributed end

# perform always and as the last $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_end_sh
```

Example 7.5 *Handling model input and output files in multi-run experiments*

7.4 Experiment Restart

When an experiment was interrupted / has failed due to any reason or in the case of partial experiment performance (cf. Section 7.5 below) it can be restarted:

- Simply restart the experiment by simenv.rst without changing any of the SimEnv files describing the experiment and/or the model. The only exception may be the values for the sub-keywords of the keyword 'experiment' in the general model configuration file <model>.cfg.
- simenv.rst has the same usage as simenv.run
- Restart can be launched in a different mode (on the login machine or under job managent system control, the latter also in other in an other job class) and sub-mode and/or on an other machine than that of the interrupted experiment.
- Dependent on the experiment log-file <model>.elog, written by the interrupted / previous new-start experiment a single model run from the complete run ensemble in the restart experiment will be
  - Performed if this run has neither a start nor a finish information in the elog-file
  - Not performed if this run has a start and a finish information in the elog-file
  - Performed anew if the run has a start information but no finish information in the elog-file.
- For the latter case a model restart shell script **<model>.rst** can be provided by the user optionally to prepare restart of this single model run (e.g., by deleting non-SimEnv temporary or output files).
  Make sure that in <model>.rst
  - **#! /bin/sh is the first line**
  - **. $SE_HOME/bin/simenv_ini_sh is performed always and as the first SimEnv dot script**
  - **. $SE_HOME/bin/simenv_end_sh is performed always and as the last SimEnv dot script**

  (cf. Tab. 5.8 on page 44 and Example 7.6 below).

Make sure that <model>.rst has execute permission by the Unix / Linux command
chmod u+x <model>.rst.
After running $SE_HOME/bin/simenv_get_run_sh the shell script variables simenv_run_int and simenv_run_char are available in <model>.rst (cf. Tab. 10.10).
Terminal output from <model>.rst is re-directed to the log-file <model>.nlog.

- Experiment restart works without standard SimEnv experiment preparation. Instead, experiment preparation files and other information from the interrupted experiment will be used.
- For a restart, the optional experiment preparation shell script **<model>.ini** will be performed only on demand. This request is specified in the configuration file <model>.cfg with the sub-keyword 'restart_ini' and its value "yes".
  For Python, Java, Matlab and GAMS models interfaced to SimEnv <model>.ini has to be performed mandatorily. Consequently, the value of restart_ini has to be set to "yes" (cf. Sections 5.5.1 and 5.7.1)
- **<model>.cfg** will be checked anew for experiment restart. Do not change for a restart any of the information related to the keyword 'model' in <model>.cfg.
- Minutes of the restarted experiment will be appended to the log-files <model>.mlog, <model>.nlog, and <model>.elog, respectively from the interrupted experiment.
- Restart can be applied to an experiment several times successively.
- Experiment restart can be performed also as a partial experiment, independently on the partial status of the original model
- Experiment re-start is not possible for the experiment type optimization.

For the model world_sh (cf. Example 15.10 on page 179) the following contents could be defined for the restart shell script world_sh.rst:

```
#! /bin/sh

# perform always and as the first $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_ini_sh

# get run number
. $SE_HOME/bin/simenv_get_run_sh

# remove all files from the temporary directory and the directory itself
if test -d run$simenv_run_char
then
   rm -fR run$simenv_run_char
fi

# perform always and as the last $SE_HOME/bin/simenv_*_sh dot script:
. $SE_HOME/bin/simenv_end_sh
```

Example file: world_sh.rst

Example 7.6 *Shell script <model>.rst to prepare model performance during experiment restart*

7.5 Experiment Partial Performance

- SimEnv enables to perform an experiment partially by performing only a run slice out of the whole run ensemble.
- Therefor assign appropriate run numbers to the corresponding sub-keywords 'begin_run' / 'end_run' / 'include_runs' / 'exclude_runs' in <model>.cfg (check Sect. 10.1).
- A partial experiment performance is also possible for an experiment restart.
- Experiment partial performance is not possible for the experiment type optimization and for a Monte Carlo analysis with a stopping function.
- For more information check Fig. 7.1.

7.6 Experiment Related User Shell Scripts and Files

Tab. 7.1 *Experiment related user shell scripts and files*

Shell script / file	Explanation	Used for (*)	Exist status
Shell scripts (terminal output is re-directed to <model>.nlog)		**(**)**	
<model>.run	model shell script to wrap the model executable Model interface dot scripts at shell script level simenv_*_sh can / have to be applied in <model>.run: • $SE_HOME/bin/simenv_ini_sh has to be performed always and as the first SimEnv dot script simenv_*_sh • $SE_HOME/bin/simenv_end_sh has to be performed always and as the last SimEnv dot script simenv_*_sh	S R	mandatory
<model>.rst	model shell script to prepare single model run restart for such single runs that were started but not finished during the previous experiment start / restart • $SE_HOME/bin/simenv_ini_sh has to be performed always and as the first SimEnv dot script simenv_*_sh • $SE_HOME/bin/simenv_end_sh has to be performed always and as the last SimEnv dot script simenv_*_sh • $SE_HOME/bin/simenv_get_run_sh can be used	R	optional
<model>.ini	model shell script to prepare simulation experiment additionally to standard SimEnv preparation • Experiment will be not performed if return code from this shell script is unequal 0 • For experiment re-start <model>.ini will be performed only on request	S (R)	optional, for Python, Java, Matlab and GAMS models mandatory
<model>.end	model shell script to clean up simulation experiment from non-SimEnv files	S R	optional
Files			
<model>.err <simenv_run_char>	touch such a file in <model>.run and/or in <model>.rst as an indicator to stop the complete experiment after single run <simenv_run_int> has been finished	A	optional
<model>.killed <simenv_run_char>	generated from $SE_HOME/bin/simenv_kill_process in <model>.run as an indicator that a process exceeded the specified CPU-time limit and was killed	A	optional
<model>_ [dis | par | seq]_ [aix | linux].jcf	model-specific job control file to submit an experiment in distributed, parallel and/or sequential sub-mode by the load leveler LoadL • Copy from $SE_HOME/bin/simenv_[dis | par | seq]_ [aix | linux].jcf if required	J	optional
<model>_opt_ options.txt	model-specific control and option file for experiment type optimization • Copy from $SE_HOME/bin/simenv_opt_options.txt if required	O	optional

(*): *shell script applied for*
R: ***R**estart of an experiment by simenv.rst <model>*
S: ***S**tart of an experiment by simenv.run <model>*
file applied for
A: ***A**ll experiment performance on the login machine or by submission to a job management system*
J: ***J**ob management experiment submission*
O: ***O**ptimization experiment performance*

(**): *make sure by the Unix / Linux command chmod u+x <model>.<ext> that the shell script <model>.<ext> has execute permission*

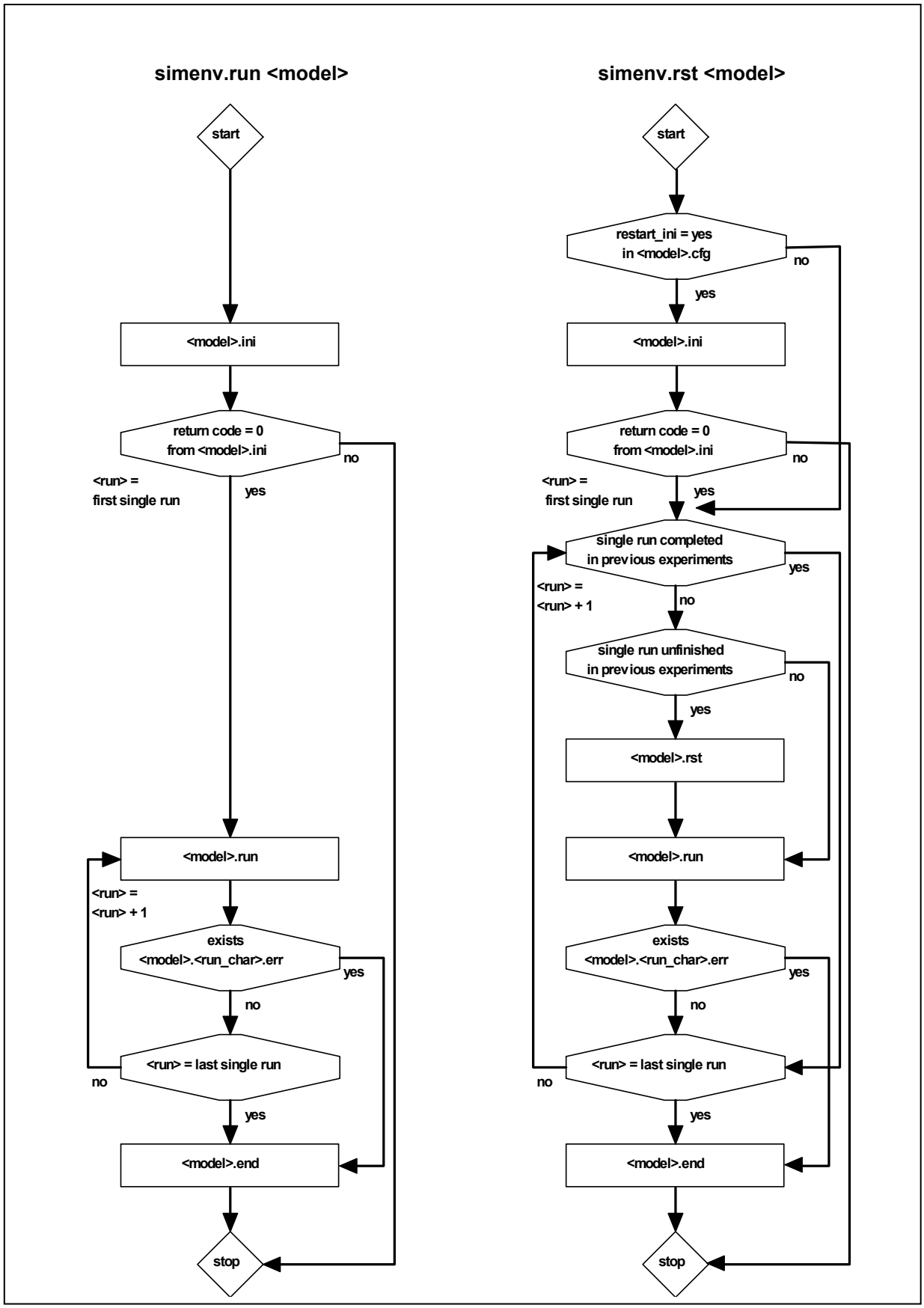


Fig. 7.1 *Flowcharts for performing simenv.run and simenv.rst*
First and last single run always refer to the corresponding settings in <model>.cfg

7.7 Saving Experiments

To save experiments for later use, e.g., by SimEnv experiment post-processing, make sure to store the files listed in Tab. 7.2:

Tab. 7.2 *SimEnv files to store for later experiment post-processing*

File name	Remark
Mandatory from the model output directory	
<model>.out[all | <simenv_run_char>].[nc | ieee]	
Mandatory from the current workspace	
<model>.cfg	do not modify the information assigned to the keyword 'model'
<model>.mdf	do not modify all information including the sequence of the model output variables and/or experiment factors
<model>.edf	
<model>.smp	
<model>.fct	for optimization and Monte Carlo experiment with stopping rule
Optional from the current workspace	
<model>.elog	
<model>.mlog	
<model>.nlog	
<model>_[dis | par | seq]_[aix | linux].jcf	
<model>.olog	for optimization experiment
<model>_opt_options.txt	for optimization experiment

8 Experiment Post-Processing

Goal of experiment post-processing is to navigate within the model / experiment output space by deriving interactively output functions / data that are to be visualized in experiment evaluation afterwards. Therefor SimEnv supplies operators that can be applied to model output and reference data. There are built-in basic and advanced operators and built-in experiment specific operators. The user can define its own private operators and easily couple them to the post-processor. Additionally, composed operators can be derived from both built-in and user-defined operators. Operator chains and recursions are possible. Macros can be defined as abbreviations for operator chains.

8.1 General Approach

8.1.1 Post-Processor Results

In SimEnv experiment post-processing post-processor results (synonym: output functions) are derived from model output of the experiment and from reference data. A post-processor result is specified by a post-processor expression, optionally prefixed by a result description and a result unit string:

<result> = { { <result_description> } { [<result_unit>] } := } <result_expression>

<result>	by the string “Enter a result” the user is asked to enter a result. Input lines with a character # as the first non-white space character are treated as comments. The experiment post-processing session is finished by entering <ret> or a sequence of white spaces instead of a result. For case sensitivity of <result> check Tab. 10.12 on page 137.
<result_description>	must not contain an apostrophe character “‘”.
<result_unit>	characters “[” and “]” belong to the syntax and are **not** part of the this document convention as defined in Tab. 1.1 Result description and/or unit together with the separator “:=” have to be specified in the first input line. The result expression itself may follow at the following input line.
<result_expression>	is a chain of SimEnv operators applied to model output variables and/or reference data. Can be continued on a new input line (continue expression:) if the current input line ends on one of the operators “+” , “-” , “*” , “/”, or “**” or on the operand separator “,” in operators. White spaces are filtered out from the result expression string, also from character arguments.

<result_description> or <result_unit> are used to describe the result in the corresponding result output file (cf. Chapter 12). For the case one of these entities is not specified SimEnv analyses the result expression: For a result expression formed without any operator or only from one operator and using exactly one model output variable and/or one experiment factor <result_description> and/or <result_unit> is copied from the corresponding information for the sub-keyword ‘descr’ in <model>.mdf (for a model output variable as an operand of this operator) and/or from <model>.edf (for an experiment factor as an operand of this operator). The only operator used in this expression must not transform the contents of the operand in general (must be invariant with respect to description and unit). For all other cases <result_description> is set to the string res_<xy> and <result_unit> is undefined.

Having a model output variable definition as in Example 5.1 on page 31 then in experiment post-processing

`abs(atmo)+3`	applies operator abs to atmo and adds 3 (multi-operator result expression) <result_description> = 'res_<xy>' <result_unit> undefined
`Energy [MWh] := abs(atmo)+3`	as above, but: <result_description> = 'Energy' <result_unit> = 'MWh'

`Energy [MWh] :=`
`abs(atmo)+`
`3`
	as above

`[MWh] := abs(atmo)+3`	as above, but: <result_description> = 'res_<xy>' <result_unit> = 'MWh'
`sign(atmo)`	applies operator sign to atmo (operator sign is not invariant w.r.t. the contents of its operand) <result_description> = 'res<xy>' <result_unit> undefined
`abs(atmo)`	applies operator abs to atmo (operator abs is invariant w.r.t. the contents of its operand) <result_description> = 'aggregated atmospheric state' (according to <model>.mdf) <result_unit> = 'without' (according to <model>.mdf)
`Energy := abs(atmo)`	applies operator abs to atmo <result_description> = 'Energy' (according to <model>.mdf) <result_unit> = 'without' (according to <model>.mdf)

Example 8.1 *Addressing results in experiment post-processing*

8.1.2 Operands

Operands in result expressions can be

- Model output variables — as defined in <model>.mdf
  In the following abbreviated by **arg**
  Example: `atmo`
- Experiment factors — as defined in <model>.edf
  In the following abbreviated by **arg**
  Example: `p1`
- Constants <int_val> or <real_val>
  In the following abbreviated by **int_arg and/or real_arg**
  Example: `12` and `-12` and `12.34` and `-1.234e+1`
- Character strings <string>, enclosed in single quotation marks
  In the following abbreviated by **char_arg**
  Example: `'tie_avg'`
- Operator results
  In the following abbreviated by **arg**
  Example: `abs(atmo)` and `atmo+3.`
- Macros — as defined in <model>.mac (cf. Section 8.7)
  Example: `equ_100yrs_m`

- Wildcard operands (cf. Section 8.8)
  Example: `&v&`

As for model output variables (cf. Section 5.1) also to each operand (with the exception of character string operands)

- Dimensionality **dim(operand)** and
  Extents **ext(operand,i)** with i=1 ,..., dim(operand)
  Coordinates **coord(operand,i)** with i=1 ,..., dim(operand)

  are assigned to. The dimensionality is the number of dimensions, an extent is related to each dimension and represents the number of elements in that dimension. Extents are always greater than 1. To each dimension a coordinate is assigned to. Coordinates have a name and from all coordinate values the coordinate is defined for a subset is assigned to the extent of the dimension of the operand. Coordinate specification for operands follows that for model output variables. For more information see Section 5.1.
- Operators transform dimensionality, dimensions, and coordinates of the their non-character operator arguments into unique dimensionality, dimensions and coordinates of the operator result (cf. Section 8.1.4).
- Consequently, the output of an operator and finally a post-processor result as a sequence of operators applied to operands also has unique dimensionality, extents and coordinates.
- Experiment factors and constants always have a dimensionality of 0.
- Operands of dimensionality 0 and character string operands do not have a coordinate assignment.

8.1.3 Model Output Variables

- A variable of dimensionality n corresponds to an n-dimensional array and is defined at an n-dimensional grid, spanned up from the coordinate values of the assigned coordinates The complete data field of a model output variable or parts of it can be addressed in experiment post-processing (see below). Dimensionality, dimensions and coordinate description of this data field is derived from the model output variable description in <model>.mdf.
- Model output variables are specified in the ASCII model output description file <model>.mdf (cf. Tab. 5.3 on page 29) by their
  - Name
  - Dimensionality
  - Extents
  - Coordinate assignment to each dimension
  - Data type (cf. Tab. 5.4 on page 30).
  - Use the service simenv.chk to check variables description in model output description file <model>.mdf
- Addressing of model output data fields or parts of it is done in experiment post-processing by corresponding model output variables names.
- For variables with a dimensionality greater than 0 it is possible to address only a part of the whole variable field by
  - Specifying for a dimension an **index range i** by
    i = <index_value$_1$> { : <index_value$_2$> }
    <index_value$_1$> ≤ <index_value$_2$>
    <index_value$_2$> = <index_value$_1$> if <index_value$_2$> is missing.
    i= stands for index addressing
  - Specifying for a dimension a **coordinate range c** by
    c = <coordinate_value$_1$> { : <coordinate_value$_2$> }
    <coordinate_value$_1$> ≤ <coordinate_value$_2$> for strictly increasing coordinate values
    <coordinate_value$_1$> ≥ <coordinate_value$_2$> for strictly decreasing coordinate values
    <coordinate_value$_1$> = <coordinate_value$_2$> if <coordinate_value$_2$> is missing
    c= stands for coordinate addressing
  - Index and coordinate ranges are separated from each other by a comma, the sequence of ranges for all dimensions is enclosed in brackets and is appended after the variable name.
  - For one variable c= and i= can be used in mixed mode for different dimensions.
    * denotes the complete range of a dimension.
    c= * is identical to i= * is identical to *

- In the general SimEnv configuration file <model>.cfg (cf. Section 10.1 on page 125) a global default for index and/or coordinate addressing is established for the whole experiment post-processing session. This global default can be overwritten locally by using c= and/or i=.

Having a model output variable definition as in Example 5.1 on page 31 then in experiment post-processing result expressions can be

`atmo`	and
`atmo(*,*,*,*)`	and
`atmo(c=*,*,i=*,*)`	and
`atmo(c=88:-88,c=-178:178,c=1:16,c=1:20)`	and
`atmo(i=1:45,i=1:90,i=1:4,i=1:20)`	and
`atmo(i=1:45,c=-178:178,*,*)`	and
`atmo(1:45,1:90,1:4,1:20)`	and (with address_default = index in model.cfg)
`atmo(1:45,c=-178:178,1:4,1:20)`	and (with address_default = index in model.cfg) all address all 45*90*4*20 values and the following holds for this addressed variable: Dimensionality = 4 Coordinates = lat , lon , level , time Extents = 45 , 90 , 4 , 20
`atmo(*,*,*,c=11:20)`	addresses all values of last 10 decades Dimensionality = 4 Coordinates = lat , lon , level , time Extents = 45 , 90 , 4 , 10
`atmo(*,*,c=1,c=1)`	addresses all values of the first decade for level 1 Dimensionality = 2 Coordinates = lat , lon Extents = 45 , 90
`atmo(c=0,*,1,i=20)`	addresses all values of level 1for the last decade at equator Dimensionality = 1 Coordinates = lon Extents = 90
`atmo(i=23,*,1,i=20)`	addresses all values of level 1for the last decade at equator Dimensionality = 1 Coordinates = lon Extents = 90
`atmo(c=0,c=2,c=1,c=20)`	addresses the value for the last decade at (lat,lon,level,time) = (0°,2°,1,20) Dimensionality = 0 Coordinates = (without) Extents = (without)
`atmo(c=0,c=1:9,c=1,c=20)`	addresses the values for the last decade at (lat,lon,level,time) = (0°,2°,1,20) and (0°,6°,1,20) Dimensionality = 1 Coordinates = lon Extents = 2
`atmo(c=0,c=1,c=1,c=20)`	error in addressing: c=1 for lon does not exist

Example file: world.post_bas

Example 8.2 *Addressing model output variables in experiment post-processing*

8.1.4 Operators

- Operators transform dimensionality, dimensions, and coordinates of the their non-character operator arguments into unique dimensionality, dimensions and coordinates of the operator result (cf. Section 8.1.2).
  There are
  - Single-argument operators that replicate dimensionality, dimensions and coordinates from the only argument to the operator result
    Example: `sin(atmo)`
  - Multi-argument operators that demand a certain relation between dimensionalities, dimensions and coordinates of their arguments
    Example: `mod(atmo(c=84:-56,*,c=1,*),bios)`
  - Operators that increase the dimensionality of the operator result and assign new coordinates to the additional dimensions (cf. Tab. 10.11) or form new coordinates from resulting factor adjustments Example: `ens(atmo)`
- SimEnv experiment post-processing operators may have two special types of arguments:
  - Character arguments char_arg:
    Only character strings enclosed in ‘ ‘ are valid as arguments. Some built-in operators (e.g., count) have a pre-defined set of valid character argument strings (e.g., for operator count strings all, def, and undef). Some built-in operators allow an empty string (e.g., behav)
    Example: `count('undef',atmo)`
    `behav(' ',atmo)`
  - Integer or real (float) constant arguments int_arg or real_arg:
    Only constants in appropriate format are valid as arguments. Model output variables of dimensionality 0 or general operands with dimensionality 0 are invalid.
    Example: `move_avg('0001','lin',3,atmo)`
    `qnt(33.333,atmo)`
  - If character and integer/real constant arguments are defined for an operator then there is always the following sequence of the operator arguments:
    { char_arg } { int_arg } { real_arg } { arg }
    Example: `hgr_l('1000','bin_mid',20,0.,0.,atmo)`
- Operators are generic with respect to the data types of their operands: Each non-character and non-constant argument can be used with operands of all defined data types (cf. Section 5.1). Internally, arguments of any type are converted to a float representation. This may lead to undefined arguments of type double in float representation.
- Results of SimEnv experiment post-processing operators are always of the type float.
- SimEnv post-processing follows the standard approach for description of operators for basic as well as advanced built-in or user-defined operators.
  Advanced built-in or user-defined operators
  - Have a unique name and a number of operands
  - The sequence of operands is enclosed in parentheses directly after the operator name
  - Operands are separated from each other by a comma.
  - Recursions of the same operator (also for user-defined operators) are possible.
    Example: `log10(min_n(3 , min_n(log10(atmo(*,*,1,c=20)) , 400) , 10*bios_g))`
- Elemental operators use the common form of notation:
  Example: `atmo_g + 345`

8.1.5 Operator Classification, Flexible Coordinate Checking

Tab. 8.1 lists for all built-in operators a classification of argument restrictions and result description that are used in the following for the explanation of built-in operators.

Tab. 8.1 *Classified argument restriction(s) / result description*
(*): for the different levels of checking a coordinate description see below

Argument restriction(s) / result description		**Argument restriction(s)**	**Result description (cf. Section 8.1.2 for syntax)**
(1)		dimensionality, extents and coordinates of the only non-character / non-constant argument arg can be arbitrary	same dimensionality, extents and coordinates as the only non-character / non-constant argument: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j
(2)	(2.1)	all non-character / non-constant arguments arg with same dimensionality, extents and coordinates **(*)**	same dimensionality, extents and coordinates as all the non-character / non-constant arguments: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j
	(2.2)	some non-character / non-constant arguments arg with same non-zero dimensionality, extents and coordinates **(*)**, all the other non-character arguments with dimensionality 0	same dimensionality, extents and coordinates as all the non-character / non-constant arguments with non-zero dimensionality: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j the 0-dimensional argument is applied to each element of the non-zero dimensional argument
(3)		dimensionality, extents and coordinates of the only non-character argument can be arbitrary	dim(res) = 0
(4)	(4.1)	all non-character / non-constant arguments with same dimensionality, extents and coordinates **(*)**	dim(res) = 0
	(4.2)	some non-character / non-constant arguments with same non-zero dimensionality, extents and coordinates **(*)**, all the other non-character / non-constant arguments with dimensionality 0	dim(res) = 0 the 0-dimensional argument is applied to each element of the non-zero dimensional argument
(5)		dimensionality, extents and coordinates of the first non-character / non-constant argument arg can be arbitrary, all the other following arguments have to have dimensionalities, extents and coordinates **(*)** of this argument or have to have dimensionality 0	same dimensionality, extents and coordinates as the first non-character / non-constant argument: dim(res) = dim(arg) ext(res,j) = ext(arg,j) for all j coord(res,j) = coord(arg,j) for all j
(6)		Only character arguments or without arguments	dim(res) = 0

The requirement for a lot of operators to have same coordinates for same dimensions may restrict application of experiment post-processing especially for hypothesis checking heavily. To enable a broader flexibility with respect to this situation a general solution is provided by SimEnv post-processing: With the sub-keyword 'coord_check' in the general configuration file <model>.cfg three different modi can be assigned globally to the SimEnv complete post-processing session:

- coord_check = strong
  To ensure for two arguments with same dimensionalities and extents to have same coordinates it is necessary that
  - Assigned coordinate values for corresponding dimensions are unique
  - Assigned coordinate names for corresponding dimensions are unique

  coord_check = strong is the default
- coord_check = weak
  To ensure for two arguments with same dimensionalities and extents to have same coordinates it is necessary that
  - Assigned coordinate values for corresponding dimensions are unique
  - Assigned coordinate names may differ.

  Coordinate description of the appropriate operator result is delivered from its first non-character / non-constant operand.
- coord_check = without
  To ensure for two arguments with same dimensionalities and extents to have same coordinates
  - Neither coordinate names nor coordinate values for corresponding dimensions are checked

  Coordinate description of the appropriate operator result is delivered from its first non-character / non-constant operand.

Check Example 8.3 for examples.

Having a model output variable definition as in Example 5.1 on page 31 then the checking rules for coordinates are applied in the following manner to operands with dimensionality 1:

Result expression	Same coordinates for coord_check =		
	strong	weak	without
`bios(*,*,*) + atmo(c=84:-56,*,c=1,*)` (same coordinate names, same coordinate values)	yes	yes	yes
`atmo_g(*) + hgr('bin_no',20,0.,0.,atmo)` (differing coordinate names, same coordinate values)	no	yes	yes
`atmo_g(c=6:16) + atmo_g(c=8:18)` (same coordinate names, differing coordinate values)	no	no	yes
`atmo_g(c=20) + atmo(c=0,c=2,c=1,c=1)` (two operands with dimensionality 0)	yes	yes	yes

While determination of coordinate information is unique for coord_check = strong,
coordinate information is determined by the first summand for coord_check = [weak | without].

Example 8.3 *Checking rules for coordinates*

8.2 Built-In Generic Standard Aggregation / Moment Operators

The generic operators in Tab. 8.2 can be applied during experiment post-processing to derive aggregations and moments from operands in different ways by appending suffixes (_n, _l, _e, without suffix) to the generic operator name or by incorporating them into the filter argument for experiment specific operators of bahavioural analysis:

Tab. 8.2 *Built-in generic standard aggregation / moment operators*

Generic aggregation and moment operator	**Meaning**
max	maximum of values
min	minimum of values
sum	sum of values
avg	arithmetic mean of values
var	variance of values
avgg	geometric mean of values
avgh	harmonic mean of values
avgw	weighted mean of values
hgr	histogram of values
count	number of values
maxprop	maximal, suffix related property of values
minprop	minimal, suffix related property of values

For more information check Sections 8.3.3 and 8.4.1.

8.3 Built-In Elemental, Basic, and Advanced Operators

8.3.1 Elemental Operators

Tab. 8.3 *Built-in elemental operators*

Name	**Meaning**	**Argument restriction(s) / result description (Tab. 8.1, page 84)**	**Argument value restriction**	**Precedence**
(	left parenthesis	-		first
)	right parenthesis	-		first
arg1 ** arg2	exponentiation	(2)	arg1 > 0	second
arg1 * arg2	multiplication	(2)		third
arg1 / arg2	division	(2)	arg2 ≠ 0	third
arg1 + arg2	addition (dyadic +)	(2)		fourth
arg1 – arg2	subtraction (dyadic -)	(2)		fourth
+ arg	identity (monadic +)	(1)		fourth
– arg	negation (monadic -)	(1)		fourth

- n-dimensional matrix algebra of built-in elemental operators is performed element by element
  Example: `atmo(*,*,1,*) * bios(*,*,*)` = “atmo(i,j,1,k) * bios(i,j,k)” for all addressed (i,j,k)
- If an argument value restriction is not fulfilled for an operand element the corresponding element of the operator result is undefined.
- For examples check Section 8.3.5.

8.3.2 Basic and Trigonometric Operators

Tab. 8.4 *Built-in basic and trigonometric operators*

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 84)	Argument value restriction	Example
Basic operators				
abs(arg)	absolute value	(1)		`abs(-3) = 3.`
dim(arg1,arg2)	positive difference	(2)		`dim(10,5) = 5.` `dim(5,10) = 0.`
exp(arg)	exponential function	(1)		`exp(1.) = 2.7183`
int(arg)	integer truncation value	(1)		`int(7.6) = 7.` `int(-7.6) = -7`
log(arg)	natural logarithm	(1)	arg > 0	`log(2.7183) = 1.`
log10(arg)	decade logarithm	(1)	arg > 0	`log10(10) = 1.`
mod(arg1,arg2)	remainder	(2)	arg2 ≠ 0	`mod(10,4) = 2.`
nint(arg)	nearest integer value	(1)		`nint(7.6) = 8.` `nint(-7.6) = -8.`
sign(arg)	sign of value	(1)		`sign(-3) = -1.` `sign(0) = 0.`
sqrt(arg)	square root	(1)	arg ≥ 0	`sqrt(4) = 2.`
Trigonometric operators				
sin(arg)	sine	(1)		`sin(0) = 0.`
cos(arg)	cosine	(1)		`cos(0) = 1.`
tan(arg)	tangent	(1)	arg ≠ π/2±n*π	`tan(0) = 0.`
cot(arg)	cotangent	(1)	arg ≠ ±n*π	`cot(1.5708) = 0.`
asin(arg)	arc sine	(1)	abs(arg) ≤ 1	`asin(0) = 0.`
acos(arg)	arc cosine	(1)	abs(arg) ≤ 1	`acos(1) = 0.`
atan(arg)	arc tangent	(1)		`atan(0) = 0.`
acot(arg)	arc cotangent	(1)		`acot(0) = 1.5708`
sinh(arg)	hyperbolic sine	(1)		`sinh(0) = 0.`
cosh(arg)	hyperbolic cosine	(1)		`cosh(0) = 1.`
tanh(arg)	hyperbolic tangent	(1)		`tanh(0) = 0.`
coth(arg)	hyperbolic cotangent	(1)	arg ≠ 0	`coth(3.1416) = 1.`

The following explanations hold for the operators in Tab. 8.4:

- **All operators** are applied to each element of the argument(s). These operators deal with an unfulfilled argument value restriction for an operand element in a way that the corresponding element of the operator result will be undefined.
- For examples check Section 8.3.5.

8.3.3 Standard Aggregation / Moment Operators

The generic standard aggregation / moment operators in Tab. 8.2 can be applied during experiment post-processing to derive aggregations and moments from operands in different ways by appending suffixes to the generic operator name:

- Appending **no suffix**:
  Aggregate the only non-character / non-constant argument
  Result is a scalar (an operator result of dimensionality 0) for all but operators hgr, minprop and maxprop.
  For operator hgr dimensionality of the result is 1, the extent is the specified number of bins for the histogram and the coordinate assigned has the name bin. Coordinate values are equidistant with 1 as the first value and an increment of 1.
  For operators minprop and maxprop dimensionality of the result is 1. For argument dimensionality greater / equal 1 extent of the result is equal to the argument dimensionality. Assigned coordinate name is index. Coordinate values are equidistant with 1 as the first value and an increment of 1. For argument dimensionality 0 result dimensionality is 0.
- Appending **suffix _n** (for n arguments)
  Aggregate an arbitrary number of non-character / non-constant arguments with argument restriction(s) / result description according to (2) in Tab. 8.1 on page 84 element by element
  Currently, only operators min_n and max_n are implemented.
  Result has same dimensionality, extents and coordinates as the arguments
- Appending **suffix _l** (for loop)
  Aggregate the only non-character / non-constant argument separately for selected dimensions. Dimensions to select are described by an additional loop character argument (corresponds to the group by-clause of the standard query language SQL of relational database management systems).
  Result has a lower dimensionality as the only non-character argument according to the loop character argument.
  For operator hgr_l, dimensionality is increased additionally by one, the additional extent is the specified number of bins for the histogram and the additional coordinate assigned to has the name bin. Coordinate values are equidistant with 1 as the first value and an increment of 1.
  For operators minprop_l and maxprop_l dimensionality is modified in the same manner like for operators minprop and maxprop, respectively.
- For **examples** check Section 8.3.5.

Tab. 8.5 *Built-in standard aggregation / moment operators without suffix*

Aggregation and moment operator	**Argument restriction(s) / result description (Tab. 8.1, page 84)**
max(arg)	(3)
min(arg)	
sum(arg)	
avg(arg)	
var(arg)	
avgg(arg)	
avgh(arg)	
avgw(arg1, arg2)	(4.1) arg2 = weight

Aggregation and moment operator	Argument restriction(s) / result description (Tab. 8.1, page 84)
hgr(char_arg1, int_arg2, real_arg3, real_arg4, arg5)	dim(res) = 1 ext(res,dim(res)) = number of bins for char_arg1 = 'bin_no' (bin number): coord(res,dim(res)) = name = bin_no values = equidist_end 1(1) number of bins for char_arg1 = 'bin_mid' (bin mid): coord(res,dim(res)) = name = bin_mid values = equidist_end 1st bin mid (bin width) number of bins char_arg1 see above int_arg2 = number of bins: 4 ≤ int_arg2 ≤ number_of_values or = 0: automatic determination: number of bins = max(4,number_of_values_of_arg5/10) real_arg3 left bin bound for bin number 1 real_arg4 right bin bound for bin number int_arg2 real_arg3 = real_arg4 = 0.: determine bounds by min(arg5) and max(arg5) min(arg5) = max(arg5): all result values are undefined
count(char_arg1, arg2)	(3) char_arg1 = [all | def | undef]
maxprop(arg)	dim(res) = 1 for dim(arg) > 1 ext(res,1) = dim(arg) dim(res) = 0 else return the index of that element of arg where the extreme is reached the first time according to the processing sequence of the argument field arg by the Fortran storage model (cf. Section 15.7 - Glossary).
minprop(arg)	

Tab. 8.6 *Built-in standard aggregation / moment operators with suffix _n*

Aggregation and moment operator	Argument restriction(s) / result description (Tab. 8.1, page 84)
max_n(arg1 ,..., argn)	(4)
min_n(arg1 ,..., argn)	
maxprop_n(arg1 ,..., argn)	(4) return per result element the argument position (1 ,..., n) where the extreme is reached the first time. Processing sequence starts with arg1.
minprop_n(arg1 ,..., argn)	

Tab. 8.7 *Built-in standard aggregation / moment operators with suffix _l*

Aggregation and moment operator	**Argument restriction(s) / result description**	
min_l(char_arg1,arg2)	dim(argi) > 1 ext(argi) = arbitrary dim(res), ext(res,i) according to char_arg1 and argi	
max_l(char_arg1,arg2)		
sum_l(char_arg1,arg2)		
avg_l(char_arg1,arg2)		
var_l(char_arg1,arg2)		
avgg_l(char_arg1,arg2)		
avgh_l(char_arg1,arg2)		
avgw_l(char_arg1, arg2, arg3)		dim(arg2) = dim(arg3) ext(arg2,i) = ext(arg3,i) arg3 = weight
hgr_l(char_arg1, char_arg2, int_arg3, real_arg4, real_arg5, arg6)		dim(res) = 1 + dim(res) of all other operators ext(res,dim(res)) = number of bins for char_arg2 = 'bin_no' (bin number): coord(res,dim(res)) = name = bin_no values = equidist_end 1(1) number of bins for char_arg2 = 'bin_mid' (bin mid): coord(res,dim(res)) = name = bin_mid values = equidist_end 1st bin mid (bin width) number of bins char_arg2 see above int_arg3 number of bins 4 ≤ int_arg3 ≤ number_of_values_of_arg6 or 0: automatic determination = max(4,number_of_values/10) real_arg4 left bin bound for bin number 1 real_arg5 right bin bound for bin number int_arg3 real_arg4 = real_arg5 = 0.: determine bounds by min(arg6) and max(arg6) min(arg6) = max(arg6): all result values are undefined
count_l(char_arg1, char_arg2, arg3)		char_arg2 = [all | def | undef]
minprop_l(char_arg1, arg2)	as above, but: dim(res) is increased by 1 w.r.t. above.	return the indices of those elements of arg2 where the extreme is reached the first time according to char_arg1 and to a For-

Aggregation and moment operator	Argument restriction(s) / result description	
maxprop_l(char_arg1, arg2)	ext(res,dim(res)) = dim(arg2) coord(res,dim(res)): name = index values = equidist_end 1(1)”n”	tran-like processing sequence / storage model (cf. Section 15.7 - Glossary) of the argument field arg2.

The loop character argument char_arg1 is characterised as follows:

- The length of the string is equal to the dimensionality of the non-character argument
- The string consists of 0 and 1
- 0 at position n means: aggregate over the corresponding dimension n of the argument
- 1 at position n means: do not aggregate over the corresponding dimension n of the argument
- Loop character arguments completely formed of 0 or 1 are forbidden

8.3.4 Advanced Operators

Tab. 8.8 *Built-in advanced operators*

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 84)	Argument value restriction	Example

classify(
int_arg1,
real_arg2,
real_arg3,
arg4)
	classify arg4 into int_arg1 classes;
	potentially restrict classi-fication to interval (real_arg2 , real_arg3).
		(1)
		dim(arg4) > 0
		int_arg1 = number of classes
		2 ≤ int_arg1 ≤ number of values of arg4
		= 0: automatic determina tion: number of classes = max(2,number ofvalues/10)
		real_arg2 = minimum bound for values in class # 1
		real_arg3 = maximum bound for values in class # int_arg1
		arg2 = 0. and arg3 = 0.: automatic bound determination
				`classify(`
				`10,`
				`0.,`
				`0.,`
				`atmo)`

clip(
char_arg1,
arg2)
	clip arg2 according to char_arg1	dim(arg2) > 0
		dim(res), ext(res,i) depend on char_arg1 and arg2
		char_arg1 = clip range
				`clip(`
				`‘0,*,1,10’,`
				`atmo)`

cumul(
char_arg1,
arg2)
	cumulate arg2 according to char_arg1	(1)
		dim(arg2) > 0
		char_arg1 = cumulation indicator per dimension
				`cumul(`
				`‘0001’,`
				`atmo)`

flip(
char_arg1,
arg2)
	flip arg2 according to char_arg1	(1), but coordinates are also flipped
		dim(arg2) > 0
		char_arg1 = flip indicator per dimension
				`flip(`
				`‘0001’,`
				`atmo)`

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 84)	Argument value restriction	Example

get_data(
char_arg1,
char_arg2,
char_arg3,
arg4)
	get data from an external file	dimensionality, extents and coordinates according to char_arg3 and char_arg4
		char_arg1 = data file format
		= [netcdf | ascii]
		char_arg2 = data file name
		char_arg3 = coordinate specification / transformation file name
		arg4 = variable to get from the data file
				`get_data(`
				`'nc',`
				`'data.nc',`
				`'data.def',`
				`variable)`

get_experiment(
char_arg1,
char_arg2,
char_arg3,
arg4)
	include an other experiment	(1) but
		coordinates according to char_arg3
		char_arg1 = experiment directory
		char_arg2 = model experimented with
		char_arg3 = coordinate transformation file
		arg4 = result from the other experiment
				`get_experiment(`
				`'mod_res',`
				`'mod',`
				`'mod.ctf',`
				`avg(atmo)-400)`

get_table_fct(
char_arg1,
arg2)
	apply table function with linear interpolation of table char_arg1 to arg2	(1)
		char_arg1 = file name
				`get_table_fct(`
				`'table.usr',`
				`atmo)`

if(
char_arg1,
arg2,
arg3,
arg4)
	conditional if-construct	(5)
		char_arg1 = comparison operator
		arg2 = comparator
		arg3, arg4 = new assignments
				`if(`
				`'<',`
				`atmo,`
				`400,`
				`atmo)`

mask(
char_arg1,
arg2,
arg3)
	mask values of arg2 (set them undefined) by comparing arg2 and arg3 using operator char_arg1	(5)
		char_arg1 = comparison operator
				`mask(`
				`'<',`
				`atmo,`
				`400)`

matmul(
arg1,
arg2)
	matrix multiplication	dim(arg1) = dim(arg2) = dim(res) = 2
		ext(res,i) according to matrix multiplication rules
				`matmul(`
				`atmo(*,*,1,1),`
				`transpose('21',`
				`atmo(*,*,1,1)))`

move_avg(
char_arg1,
char_arg2,
int_arg3,
arg4)
	moving average of arg4	(1)
		dim(arg4) > 0
		char_arg1 = moving average sequence per dimension
		char_arg2 = average type
		= lin: linear
		exp: exponential
		int_arg3 = running length for average
		int_arg3 > 1
		int_arg3 = 0:
		automatic determination:
		= max(3, ext(arg4,i)/20.
				`move_avg(`
				`'001',`
				`'lin',`
				`0,`
				`atmo)`

rank(
char_arg1,
arg2)
	assign rank numbers to arg2 according to ranking type argument char_arg1	(1)
		dim(arg2) > 0
		arg1 = ranking type
		[tie_plain | tie_min | tie_avg]
				`rank(`
				`'tie_avg',`
				`atmo)`

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 84)	Argument value restriction	Example

regrid(
char_arg1,
arg2)
	assign completely or partially new coordinates to arg2	(1), but
		coordinates according to char_arg1
		char_arg1 = file how to transform coordinates of arg2
		arg2 result to transform coordinates
				`regrid(`
				`'mod.ctf',`
				`atmo_g-13)`

run(
char_arg1,
arg2)
	values of arg 2 for the selected single run number explicitly or implicitly coded in char_arg1	(1)
		char_arg1 = run number selection for all experiment types:
		= <run_number>
		0 ≤ char_arg1 ≤ number_of_runs
		addit. for behavioural and local sensitivity analysis:
		= <filter argument>
		same as filter argument of operator behav, (cf. Sections 8.4.3 and 8.4.4)
				`run(`
				`'0',`
				`atmo)`
				`run(`
				`'sel_t(p1(4))',`
				`atmo)`

run_info(char_arg)	number of the current single run and/or total number of runs of the experiment	(6) char_arg1 = run information type = run_nr for current run number = nr_of_runs for number of runs of the experiment		`run_info(` `'run_nr')`

transpose(
char_arg1,
arg2)
	transpose arg2 according to sequence in char_arg1	dim(arg2) > 1
		dim(res) = dim(arg2)
		ext(res,i) = ext(arg2,j) (re-sorted)
		char_arg1 = transpose sequence
				`transpose(`
				`'3142',`
				`atmo)`

undef( )	undefined value	(6)		`undef()`

The following explanations hold for the operators in Tab. 8.8:

- **All operators but experiment and matmul** are applied to each element of the argument(s). These operators deal with an unfulfilled argument value restriction for an operand element in a way that the corresponding element of the operator result will be undefined.

- The **operator classify** transforms the values of an operand arg4 that has dimensionality > 0 into the class numbers 1 ,..., int_arg1 of int_arg1 classes. Classes are assumed to be equidistant.
  If both arguments real_arg2 and real_arg3 are 0. then min(arg4) forms the lower boundary of class number 1 and max(arg4) forms the upper boundary of class number int_arg1. For min(arg4) = max(arg4) all result values of the operator classify are undefined.
  For real_arg2 ≠ 0. or real_arg3 ≠ 0 real_arg2 and real_arg3 are used as boundaries for the classification and all of those result values are undefined where values of argument arg4 are outside the specified boundary range.

- The **operator clip** clips an operand arg2 that has dimensionality > 0. The portion to clip from the operand arg2 is described by the argument char_arg1. The argument char_arg1 uses syntax for model output variable addressing (cf. Section 8.1.3 on page 81). Note, that for all dimensions of argument arg2 lower bound index is 1. This applies also to model output variables where the lower bound index is unequal 1 in the model output description file. In general, extents differ between the result of the operator

clip and the argument arg2. Clip reduces the dimensionality of the result with respect to the argument arg2 to clip if the portion to be clipped is limited to one value for at least one dimension.
A character argument char_arg1 = '* ,..., *' results for operator clip in the identity of argument arg2.

- The **operator cumul** cumulates an operand arg2 that has dimensionality > 0. Cumulation is performed for all values of the argument arg2 from the first addressed index position up to the current index position. With the character argument char_arg1 those dimensions are identified that are to be cumulated. Character 1 at position i means cumulation across dimension i while a 0 stands for no accumulation. cumul('0...0',arg2) results in the identity to arg2.

- The **operator flip** enables flipping of variable fields. For a one-dimensional field (a vector) flip changes the value of the first index position with the value of the last position, the value of the second position with that of the last but one position, etc. With the character argument char_arg1 these dimensions are identified that are due to flip. Character 1 at position i means flipping also for dimension i while a 0 stands for no flipping at this dimension. Flipping includes adaptation of coordinates and the assigned grid. flip('0...0',arg2) results in the identity to arg2.

- With the **operator get_data** data from external files can be included in post-processing. Character argument char_arg1 specifies the data file format. Character argument char_arg2 addresses the data file. Character argument char_arg3 is used to define or transform structure information and coordinates from the data file. Argument arg4 holds the variable that is to be extracted from the data file. For restrictions in the path to the directory of the character arguments char_arg2 and char_arg3 check Tab. 11.3. Currently, ASCII and NetCDF files are supported (char_arg1 = ['ascii' | 'netcdf']).
For ASCII data files the file syntax rules from Section 11.3 are valid. Since the ASCII data file itself does not come with any structure and coordinate information the character agument char_arg3 specifies this information. For ASCII data files this argument is a mandatory one. It follows the same rules as for any coordinate transformation file (cf. Section 11.2). Keywords 'general', 'assign', and 'coordinate' and the appropriate sub-keywords from Tab. 11.5 can be used to structure the data file and to assign coordinates and coordinate values. Consequently, the keyword 'modify' is not allowed. See Example 8.4 for more information. For ASCII files it is assumed that the file holds only the values for one variable in a sequence according to the Fortran storage model (cf. Section 15.7 – Glossary). For ASCII files argument arg4 is only a dummy placeholder.
For NetCDF files argument 4 addresses the variable name to extract from the data file. The character argument char_arg3 is an optional argument. Unlike for ASCII data files, the keyword 'modify' is allowed.

Having a model output variable definition as in Example 5.1 on page 31 and assuming

a data file **data.asc** as

```
# data file with 6 values
10 , 20 , 30
40 , 50 , 60
```

and a file **data.def** to define data structure and coordinates as

```
general    descr              structure for data.asc
# assign as second dimension coordinate time
# (already defined in world_*.mdf)
assign     2                  coord              time
assign     2                  coord_extent       11:13
# assign as first dimension a new coordinate new_coord
assign     1                  coord              new_coord
assign     1                  coord_extent       100:110
coordinate new_coord          values             list 100,110
```

then
`get_data('ascii','data.asc','data.def',dummy)`
has Dimensionality = 2
Coordinates = new_coord , time
Extents = 2 , 3

and the result of this operator is a 2 x 3 matrix

```
10   30   50
20   40   60
```

To get same dimensionality, coordinates and extents but result values as the "original matrix" in data.asc
- exchange coordinate numbers in data.def: 1 by 2 and 2 by 1 and
- apply `transpose('21',get_data('ascii','data.asc','data.def',dummy))`
It has Dimensionality = 2
Coordinates = new_coord , time
Extents = 2 , 3

and the result of this operator chain is a 2 x 3 matrix

```
10   20   30
40   50   60
```

Example 8.4 *Experiment post-processing operator get_data and coordinate transformation file*

- The **operator get_experiment** is to access to external SimEnv model output from the same or an other model performed with the same or another experiment type and stored in the same or in an other model output format. Model output variables can differ from that used for the current model. Use for the experiment directory char_arg1 always that workspace the external experiment was started from. The external experiment is always post-processed completely over all single runs. Argument char_arg3 is the coordinate transformation file. It can be used to transform coordinates from the external result for usage in the current result of the current experiment. If no coordinate transformation file is to be used argument char_arg3 is empty (' '). If after potential application of a coordinate transformation file the imported result has same coordinate names as defined in the original experiment coordinate descriptions are checked against each other, otherwise coordinate descriptions are imported from the external into the original experiment. For syntax of coordinate transformation files check Section 11.2. For restrictions in the path to the directory of the character arguments char_arg1 and char_arg3 check Tab. 11.3.
  Attention:
  Make sure
  - no SimEnv service is running from the directory char_arg1 of the external experiment before applying this operator
  - to have full access permissions to the experiment directory char_arg1
  - the experiment directory char_arg1 differs from the current workspace

  In the experiment directory a file simenv_get_experiment.exc is used to exchange information between the external and the current experiment.

- With the **operator get_table_fct** a table function char_arg1 is applied to each element of the operand arg2. If necessary, table values are interpolated linearly. Outside the definition range of the table function the first and/or the last table value is used. File char_arg1 has to hold the table function and must be an ASCII file with two columns: The first column of each line is the argument value x associated with the elements of the operand arg2, the second column is the function value f(x) of the table associated with the elements of the operator result. Argument values x have to be ordered in a strictly increasing manner. Syntax rules for comments and separators in the table function file are the same as for user defined files (cf. Section 11.3). For restrictions in the path to the directory of the character argument char_arg1 see Tab. 11.3. Check the table function world.dat_tab in the example directory $SE_HOME/exa of SimEnv for more information.

- The **operator if** supplies a general conditional if-construct. It operates for each element of the operand arg2 in the following way:

```
if ( condition(char_arg1,arg2) ) then
        res=arg3
else
        res=arg4
endif
```

with

condition(char_arg1,arg2):		
	arg2 < 0	(char_arg1 = '<')
	arg2 ≤ 0	(char_arg1 = '<=')
	arg2 > 0	(char_arg1 = '>')
	arg2 ≥ 0	(char_arg1 = '>=')
	arg2 = 0	(char_arg1 = '==')
	arg2 ≠ 0	(char_arg1 = '!=')
	arg2 defined	(char_arg1 = 'def')
	arg2 undefined	(char_arg1 = 'undef')

- The **operator mask** supplies a method to mask (to set undefined) values. It operates for each element of the operand arg2 in the following way:

```
if ( condition(char_arg1,arg2,arg3) ) then
        res=undef( )
else
        res=arg2
endif
```

with

condition(char_arg1,arg2,arg3):		
	arg2 < arg3	(char_arg1 = '<')
	arg2 ≤ arg3	(char_arg1 = '<=')
	arg2 > arg3	(char_arg1 = '>')
	arg2 ≥ arg3	(char_arg1 = '>=')
	arg2 = arg3	(char_arg1 = '==')
	arg2 ≠ arg3	(char_arg1 = '!=')

- The **operator matmul** performs a simple matrix multiplication for 2-dimensional arguments arg1 and arg2.

- The **operator move_avg** performs a moving average operation successively for selected dimensions of the argument arg4.
  For a vector (a_1 , a_2 ,..., a_{len}) the moving average of running length rl is a vector (ma_1 , ma_2 ,..., ma_{len}) with elements

$$ma_i = \frac{1}{\sum_{j=\max(1,i-rl+1)}^{i} w_{ij}} \cdot \sum_{j=\max(1,i-rl+1)}^{i} w_{ij} \cdot a_j$$

  where w_{ij} are weights. Value ma_i is averaged from the rl values a_i , a_{i-1} ,..., a_{i-rl+1}. Accordingly, the first rl-1 values ma_1, ma_2 ,..., ma_{rl-1} are averaged from less than rl values.

  For the linear moving average the weights are $w_{ij} = 1$ and $\sum_{j=\max(1,i-rl+1)}^{i} w_{ij} = \min(rl,i)$,

  for the exponential moving average the weights are $w_{ij} = e^{-\frac{i-j}{rl}}$.

  While the moving average is normally applied to time-dependent one-dimensional data vectors the operator move_avg allows processing of multi-dimensional data fields in a general and succesive manner. For example, if arg4 is the three-dimensional variable bios(1:lat,1:lon,1:time) then the linear moving average could be applied to the dimension time successively for all combinations of lat and lon. This means that (lat1 = 1 ,..., lat) * (lon1 = 1 ,..., lon) = lat*lon moving averages will be performed for the vector

  (bios(lat1,lon1,1) , bios(lat1,lon1,2) ,..., bios(lat1,lon1,time)).

Afterwards this moving averaged temporary result tmp could be moving averaged for all values of lat: (lon1 = 1 ,..., lon) * (time1 = 1 ,..., time) = lon*time moving averages will be performed for the vector

(tmp(1,lon1,time1) , tmp(2,lon1,time1) ,..., tmp(lat,lon1,time1)).

The operator that allows for this double averaging would have the arguments

move_arg('201' , 'lin' , 0 , bios) .

The character argument char_arg1 supplies those dimensions that are to be involved in the moving average operation. If the n-th digit of char_arg1 is a digit > 0 then the moving average for dimension n of argument arg4 is performed at position number "digit" (i.e. after performing moving averages for those dimensions that correspond to digits smaller than the current one). If the n-th digit of arg1 is 0 then the moving average for the dimension n of arg4 will not be performed.
Keep in mind that the sequence of moving averages for single coordinates influences the result of the operator.

- The **operator rank** transforms all values of the operand arg2 that has dimensionality > 0 into their ranks. Small values get low ranks, large values get high ranks. The smallest rank is 1. Character argument char_arg1 determines how to rank ties, i.e., values arg 21 and arg22 of arg2 that are identical or have a maximum relative difference of (abs(arg21-arg22)/arg21) < 10^{-6}):
  Assume an argument arg2 with 6 values (4., 2., 4., 4., 4., 8.).

Then	char_arg1 = 'tie_plain'	returns ranks	(2 , 1 , 2 , 2 , 2 , 3) four times rank 2; next rank is 3, does not take into account the number of identical values
	char_arg1 = 'tie_min'	returns ranks	(2 , 1 , 2 , 2 , 2 , 6) four times rank 2; next rank is 6, takes into account the number of identical values
	char_arg1 = 'tie_avg'	returns ranks	(3.5 , 1 , 3.5 , 3.5 , 3.5 , 6) four times mean rank 3.5 = (2+3+4+5)/4; next rank is 6, takes into account number of identical values

- The **operator regrid** can be used to assign new coordinates to argument arg2. Character argument char_arg1 is the name of the coordinate transformation file that holds the information how to transform the coordinates. The keyword 'modify' and the corresponding sub-keywords are not allowed. For syntax of coordinate transformation files check Section 11.2. For restrictions in the path to the directory of the character arguments char_arg1 check Tab. 11.3.

- The **operator run** selects a single run from the run ensemble. The operator run must not contain experiment specific (multi-run) operators as operands, since these operators may refer to the operator run. Additionally, run must not contain itself as an argument.
  The character argument char_arg1 can hold the run number string explicitly. An explicit run number string in character argument char_arg1 is allowed for all experiment types. Additionally, for behavioural and local sensitivity analysis a run number unequal 0 can be selected implicitly by applying a filter of the corresponding operators (cf. Sections 8.4.3 and 8.4.4) as char_arg1 of the operator run.
  The file <model>.smp holds the sampled factor values to be adjusted by the default values for the current experiment. Run number n corresponds to record number n+1 of this file. Single run number 0 corresponds to the default single run 0. For more information on <model>.smp check Section 6.1 on page 53. For examples see Example 8.7 and Example 8.9.

- The **operator run_info** returns for the character argument 'run_nr' the run number of the current single run of the experiment. For the character argument 'nr_of_runs' the number of performed single runs of the current post-processed experiment without the run number 0 is returned.

- The **operator transpose** enables to transpose an operand that has a dimensionality > 1. Sequence of extents of the transposed result is described by character argument char_arg1: It consists of digits 1 ,..., dim(arg2) where the digit sequence corresponds to the re-ordered sequence of the operator result extents.
  A character argument char_arg1 = '123...' results for the operator transpose in the identity of argument arg2.

- The **operator undef** supplies a 0-dimensional result as undefined. This operator can be used as an argument for the if-operator.

- For **examples** of the described operators check Section 8.3.5.

8.3.5 Examples

Having a model output variable definition as in Example 5.1 on page 31 and
assuming address_default=coordinate in world_*.cfg then in experiment post-processing

```
atmo_g+2*atmo_g                 value of result 3*atmo_g
                                Dimensionality = 1
                                Coordinates = time
                                Extents = 20
sqrt(atmo_g)                    square root of atmo_g
                                Dimensionality = 1
                                Coordinates = time
                                Extents = 20
clip('i=23,*,1,19:20',atmo)     last two decades for level 1 at equator
                                equivalent with atmo(i=23,*,1,19:20)
                                Dimensionality = 2
                                Coordinates = lon , time
                                Extents = 90 , 2
atmo - get_experiment('./other_dir','other_model',' ',atmo)
                                Difference for atmo between the current experiment and
                                another model other_model, located in directory ./other_dir
                                without application of a coordinate transformation file
                                Dimensionality = 4
                                Coordinates = lat , lon , level , time
                                Extents = according to definition of atmo in other_model
get_table_fct('world.dat_tab',atmo)
                                Operator table_fct with table world.dat_tab applied to
                                each element of atmo
                                Dimensionality = 4
                                Coordinates = lat , lon , level , time
                                Extents = 45 , 90 , 4 , 20
if('<',atmo-10,10,atmo)         maximum from atmo and 10 for each element of atmo
                                equivalent with max_n(atmo,10)
                                Dimensionality = 4
                                Coordinates = lat , lon , level , time
                                Extents = 45 , 90 , 4 , 20
avg(atmo(*,*,*,19:20))          global all-level mean over the last two decades
                                Dimensionality = 0
                                Coordinates = (without)
                                Extents = (without)
maxprop(atmo)                   indices of this element of atmo where the maximum of atmo
                                is reached the first time
                                Dimensionality = 1
                                Coordinates = index
                                Extents=4
min_n(atmo(84:-56,*,1,19:20),10.)
                                minimum per grid cell for level 1 without polar regions
                                for the last two decades from atmo and 10
                                Dimensionality = 3
                                Coordinates = lat , lon , time
                                Extents = 36 , 90 , 2
```

```
min_l('10',atmo(20:-20,*,1,20))
                                zonal tropical minima of atmo for the last decade and
                                level 1
                                Dimensionality = 1
                                Coordinates = lat
                                Extents = 11
minprop_l('10',atmo(20:-20,*,1,20))
                                zonal tropical indices of those elements of
                                atmo for the last decade and level 1 where the minimum is
                                reached the first time
                                Dimensionality = 2
                                Coordinates = lat , index
                                Extents = 11 , 2
hgr_l('10','bin_no',8,0.,0.,atmo(20:-20,*,1,20))
                                zonal tropical histograms with 8 bins of atmo for the
                                last decade and level 1. Bin bound extremes are deviated
                                from the values of atmo
                                Dimensionality = 2
                                Coordinates = lat , bin_no
                                Extents = 11 , 8
avg_l('100',min_l('1011',atmo(20:-20,*,*,*)))
                                temporally averaged all-level zonal tropical minima
                                Dimensionality = 1
                                Coordinates = lat
                                Extents = 11
                                                        Example file: world.post_adv
```

Example 8.5 *Experiment post-processing with advanced operators*

8.4 Built-In Experiment Specific Operators

- Experiment specific operators are to navigate and process in the experiment space.
- Experiment specific operators must not be applied recursively.
- Addressing a variable within an experiment specific operator normally results in application of the operator on the whole run ensemble or parts of it and in aggregating across the run ensemble according to the operator.
- Addressing a variable outside an experiment specific operator results in application of the basic, advanced and/or user-defined operator on the variable for the default run number 0 of the experiment.
- If the dimensionality of an operator result is higher than that of one of its operands the additional dimensions of the result are appended to the dimensions of the operand. Examples for such operators are ens (for Monte Carlo analysis post-processing) and behav (for certain constellations of behavioural analysis post-processing).

8.4.1 Standard Aggregation / Moment Operators

Tab. 8.9 summarises multi-run standard aggregation / moment operators for behavioural analysis, Monte Carlo analysis and optimization. They work on the whole run ensemble (for Monte Carlo analysis and optimization) or parts of it (for certain constellations of behavioural analysis post-processing). They are used with suffix _e for Monte Carlo analysis and optimization and without any suffix for behavioural analysis. For a definition of these operators check Tab. 8.2 on page 86.

Tab. 8.9 *Multi-run standard aggregation / moment operators*

Aggregation and moment operator	**Argument restriction(s) / result description (Tab. 8.1, page 84)**
min(arg)	(1)
max(arg)	
sum(arg)	
avg(arg)	
var(arg)	
avgg(arg)	
avgh(arg)	
avgw(arg1, arg2)	(2.1) arg2 = weight
hgr(char_arg1, int_arg2, real_arg3, real_arg4, arg5) (heuristic probability density function)	dim(res) =dim(arg2)+1 ext(res,dim(res)) = number of bins for char_arg1 = 'bin_no' (bin number): coord(res,dim(res)) = name = bin_no values = equidist_end 1(1) number of bins for char_arg1 = 'bin_mid' (bin mid): coord(res,dim(res)) = name = bin_mid values = equidist_end 1st bin mid (bin width) number of bins char_arg1 see above int_arg2 = number of bins 4 ≤ int_arg2 ≤ number_of_runs or 0: automatic determination = max(4,number_of_runs/10) real_arg3 left bin bound for bin number 1 real_arg4 right bin bound for bin number arg2 real_arg3 = real_arg4 = 0.: determine bounds by min(ens(arg5)) and max(ens(arg5)) min(ens(arg5)) = max(ens(arg5)): all result values are undefined
count(char_arg1, arg2)	(1) arg1 = [all | def | undef]
minprop(arg)	(1) return the run number where the extreme is reached the first time. Processing sequence starts with run number 1.
maxprop(arg)	

8.4.2 Global Sensitivity Analysis

Tab. 8.10 *Experiment specific operator for global sensitivity analysis*

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
morris(arg)	get global sensitivity measures for argument arg	dim(res) = dim(arg) + 2 ext(res,dim(res)-1) = number_of_factors coord(res,dim(res)-1): name = factor_sequ values = equidist_end 1(1) number_of_factors ext(res,dim(res)) = 2 coord(res,dim(res)) = name = stat_measure values = equidist_end 4(1)5	
same as for Monte Carlo analysis	see Section 8.4.5	see Section 8.4.5	see Section 8.4.5

The operator morris appends two additional dimensions to the to dimensionality of its argument. The first corresponds to the number of factors and the second to the derived statistical measures. According to the coordinate values as described above the second additional dimension has the extent 2 and according to Tab. 10.11 the first index of this dimension holds the averages μ^{abs} and the second index the variances σ to describe the importance of the corresponding factors.

Additionally, this experiment type allows to post-process the whole run ensemble as a Monte Carlo analysis. Keep in mind that the factors do not follow a pre-defined distribution.

Having a model output variable definition as in Example 5.1 on page 31.
Assume the experiment description file (2) from Tab. 6.3 on page 55
then in result-processing

```
morris(max(atmo))
```
importance measures μ^{abs} and σ
for max(atmo) for the four defined factors
Dimensionality = 2
Coordinates = factor_sequ , stat_measure
Extents = 4 , 2

```
rank('tie_plain',-clip('*,i=1',morris(max(atmo))))
```
ranks the importance measure μ^{abs}
(rank 1 for the most important factor)
for max(atmo) for the four defined factors
Dimensionality = 1
Coordinates = factor_sequ
Extents = 4

Example file: world.post_h

Example 8.6 *Experiment post-processing operators for local sensitivity analysis*

8.4.3 Behavioural Analysis

There is only one experiment specific operator for behavioural analysis. With this operator behav

- A single run can be selected from the run ensemble
- The complete run ensemble can be addressed
- Sub-spaces from the experiment space can be addressed and
- Sub-spaces can be projected by aggregation and moment operators

dependent on the way the experiment factor space was to be scanned according to the sub-keyword 'comb' in the experiment description file.

To show the power of the operator behav the simple experiment layouts as described in Fig. 4.4 on page 18 are used as examples.

- With the operator behav it is possible to address for any operand a single run out of the run ensemble by fixing values of experiment factors p1 and p2 (for Fig. 4.4 (a)), a value of the parallel factors p1 or p2 (for Fig. 4.4 (b)), and values of factors p3 and p1 or p2 (for Fig. 4.4 (c)). Dimensionality and extents of the operator result is the same as that of the operand.
- Without any selection in the factor experiment space (p1,p2) and/or (p1,p2,p3) the dimensionality of the operator result is formed from the dimensionality of the operand enlarged by the dimensionality of the experiment space. Two additional dimensions are appended to the operand for Fig. 4.4 (a), one additional dimension for Fig. 4.4 (b), and two additional dimensions for Fig. 4.4 (c). For the latter two cases it is important which of the axis p1 and p2 is used for further processing and/or output of the operator result. The extents of the appended dimensions are determined by the number of sampled values.
- As a third option it is possible to select only a sub-space out of the experiment space to append to the operand. For Fig. 4.4 (a) this could be the sub-space formed from the first until the third sampled value of p1 and all adjusted values of p2 between 3 and 7. Dimensionality of the operator result increases by 2 and extents of these additional dimensions are 3 and 2 with respect to the corresponding Example 6.3 (3a) in Section 6.3.2 on page 58.
- The operator behav also enables to aggregate operands in the experiment space. In correspondence with the example in the last bullet point for Fig. 4.4 (a) the operand could be aggregated (e.g., averaged) over the first until the third sampled value of p1 autonomously for all runs with different values of p2 and afterwards this intermediate result (that now depends only on p2) could be summed up for all adjusted values of p2 between 3 and 7. Consequently, the result has the same dimensionality as the operand of behav. Sequence of performing aggregations is important.

Tab. 8.11 *Experiment specific operator for behavioural analysis*

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
behav(char_arg1, arg2)	navigation and aggregation in the experiment space for arg2 according to char_arg1	char_arg1= selection / aggregation filter according to Tab. 8.14 dim(res) = dim(arg2) + appended dimensions according to char_arg1	
ens(arg)	whole run ensemble	dim(res) = dim(arg)+1 ext(res,dim(res)) = number_of_runs coord(res,dim(res)) = name = run values = equidist_end 1(1) number_of_runs	

Tab. 8.12 *Syntax of the filter argument 1 for operator behav*

Placeholder	Explanation
<filter>	' { <operator$_1$> {, <operator$_2$> ... {, <operator$_n$> } ... } } '
<operator>	[<select_operator> | <aggreg_operator>]
<select_operator>	sel { _<factor_val_type>} (<factor_name> { <factor_val_range> })
<aggreg_operator>	<aggreg_type> {_<factor_val_type>} (<factor_name> { <factor_val_range> })
<factor_name>	name of the experiment factor according to the experiment description file
<factor_val_type>	specification how to interpret <factor_val_range>: i as a range of position **i**ndices of factor values (always count from 1) s as a range of **s**ampled factor values <factor_smp_val> a as a range of **a**djusted factor values <factor_adj_val>
<factor_val_range>	[(<val$_1$> { : <val$_2$> }) | (*)] for <val$_2$> = <nil> : <val$_2$> = <val$_1$> * : use all values from <factor_name> <val$_i$> = <int_val$_i$> for <factor_val_type> = i <val$_i$> = <real_val$_i$> else
<aggreg_type>	an aggregation / moment operator from Tab. 8.9 on page 100. The following restrictions apply: • aggregations avgw and hgr cannot be used • aggregation count has a differing syntax: count_<factor_value_type> ([all | def | undef] , <factor_name> { <factor_value_range> }) • multiple application of minprop and/or maxprop causes senseless results

The following rules hold for the operator **behav**:

- Generally, by the filter argument arg1 those runs from the run ensemble are selected and/or aggregated (here interpreted as filtered) that are used for the formation of the result.
  Consequently, if no filter is specified all runs are used:
  `behav(' ',atmo_g)`
  The select operator has to be specified only if values are to be restricted by a corresponding factor value range.
  For the aggregation and the select operator the factor value type is redundant if the value range represents the full range of values by <factor_name> or <factor_name>(*):
  `sel(p1) = sel(p1(*)) =`
  `sel_i(p1) = sel_s(p1) = sel_a(p1) =`
  `sel_i(p1(*)) = sel_s(p1(*)) = sel_a(p1(*))`
  and all these select operators are redundant.
- The select-operator can also be applied to force a certain experiment factor to be used as a coordinate in the result of the operator behav if this factor is combined in parallel with other factors. By default, the first factor of a parallel factor sub-space as declared in the normalized (see Section 6.3.1) comb-line of the experiment description file is used in the behav-result.
- Aggregation operators reduce dimensionality of the covered experiment factor space in the behav-result. The sequence of aggregation operators in the first argument of the operator behav influences the result: Computation starts with the first aggregation operator and ends with the last:
  `avg(p1), min(p2)` normally differs from `min(p2), avg(p1)`
- An unused experiment factor in the selection and aggregation filter contributes with an additional dimension to arg2 to the result of the operator behav. The extent of this additional dimension corresponds to the number of sampled values of this factor in the experiment description file.
  A factor that is restricted by any of the select operators also contributes with an additional dimension to the result of the operator behav if the number of selected values is greater than 1. The extent of the additional dimension corresponds to the number of selected values of this factor by the select operator.
  Consequently, an empty character string arg1 forces to output the operand arg2 over the whole factor space of the experiment.

- The name of the coordinate that is assigned to an additional dimension is the name of the corresponding factor. Coordinate description and coordinate unit (cf. Section 5.1 on page 25) are associated with the corresponding information for the factor from the experiment description file.
  Coordinate values are formed from adjusted factor values. For strictly ordered factor sampled values in the experiment description file and finally for strictly ordered factor adjusted values the coordinate values are ordered accordingly in an increasing or decreasing manner. Unordered factor sampled values and finally unordered factor adjusted values are ordered in an increasing manner for coordinate usage.
  The result of the operator behav is always arranged according to ascending coordinate values for all additional dimensions.
- Independently from the declared sequence of the applied aggregation- and select-operators in argument 1 of the operator behav the factors that contribute to additional dimensions of the result of the operator behav are appended to the dimensions of the operand arg2 of behav according to the sequence they are used in the normalized (see Section 6.3.1) comb-line of the experiment description file).
  From parallel changing factors that factor is used in this sequence that is addressed explicitly or implicitly by the select-operator.
- For experiment factors that are changed in the experiment in parallel, that increase dimensionality of the result and where a select-operator is missing the first factor from this parallel sub-space in the normalized (see Section 6.3.1) comb-line is used in the result.
- For experiments that use a sample file (<model>.edf: specific comb file ...) instead of explicite sample definitions (<model.edf>: specific comb [default | <combination>]) all experiment factors are assumed to be combined in parallel.

Having a model output variable definition as in Example 5.1 on page 31 and
assuming address_default = coordinate in world_*.cfg
Assume the experiment layout in Fig. 4.4 (c) on page 18 and
the corresponding experiment description file from Example 6.3 (3c) on page 58
then in result-processing

```
behav(' ',bios(*,*,20))
```
last time step of bios dependent on (p2,p1) and p3
Dimensionality = 4
Coordinates = lat , lon , p2 , p3
Extents = 36 , 90 , 4 , 3

```
behav('sel(p2)',bios(*,*,20))
```
last time step of bios dependent on (p1,p2) and p3
Dimensionality = 4
Coordinates = lat , lon , p2 , p3
Extents = 36 , 90 , 4 , 3

```
behav('sel_a(p2(4)),sel_i(p3(1))',atmo(*,*,1,*))
```
select the single run out of the run ensemble for level 1
p2 = 4 and p3 = 3.3
Dimensionality = 3
Coordinates = lat , lon , time
Extents = 45 , 90 , 20

```
behav('sel_i(p2(1:3)),sel_s(p3(2:3))',atmo(*,*,1,20))
```
last time step of atmo for level 1 depend. on (p2,p1) and p3
use only runs for p2 = 1, 2, 3 and for p3 = 6.0, 8.4
Dimensionality = 4
Coordinates = lat , lon , p2 , p3
Extents = 45 , 90 , 3 , 2

```
behav('avg_i(p2(1:3)),sel_i(p3(2:3))',atmo(*,*,1,*))
```
mean of atmo for level 1 and for runs with p2 =1, 2, 3
for each value of p3 = 8.4, 9.9
Dimensionality = 4
Coordinates = lat , lon , time , p3
Extents = 45 , 90 , 20 , 2

```
behav('min(p2),max(p3)',avg(atmo(*,*,1,19:20)))
                                determine single minima of avg(atmo) for level 1 and the
                                last two decades for each value of p2
                                afterwards determine from that the maximum over all p3.
                                Dimensionality = 0
                                Coordinates = (without)
                                Extents = (without)
behav('max(p3),min(p2)',avg(atmo(*,*,1,19:20)))
                                Result differs normally from min(p2),max(p3)
                                (previous result expression)
behav('count(def,p3),sel_i(p2=1)',bios(*,*,20))/3
                                determine single numbers of defined values of
                                bios for last decade for runs with p2=1.
                                Result consists of values 0 (for water) and 1 (for land)
                                Dimensionality = 2
                                Coordinates = lat , lon
                                Extents = 36 , 90
behav(' ',atmo(*,*,1,20)-run('sel_i(p1(1)),sel_i(p3(3))',
      atmo(*,*,1,20)))
                                deviation of the last time step of atmo for level 1
                                from the run with p1=1, p2=1, p3=6
                                dependent on (p1,p2) and p3
                                Dimensionality = 4
                                Coordinates = lat , lon , p1 , p3
                                Extents = 45 , 90 , 4 , 3

                                                        Example file: world.post_3c
```

Example 8.7 *Experiment post-processing operator behav for behavioural analysis*

8.4.4 Local Sensitivity Analysis

Tab. 8.13 shows the experiment specific operators for local sensitivity analysis that can be used in post-processing. For a definition of these operators check Tab. 4.2 on page 19.

Tab. 8.13 *Experiment specific operators for local sensitivity analysis*

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
sens_abs(char_arg1, arg2)	absolute sensitivity measure for arg2 according to char_arg1	arg1 = selection / aggregation filter dim(res) = dim(arg2) + appended dimensions according to char_arg1	
sens_rel(char_arg1, arg2)	relative sensitivity measure for arg2 according to char_arg1		
lin_abs(char_arg1, arg2)	absolute linearity measure for arg2 according to char_arg1		
lin_rel(char_arg1, arg2)	relative linearity measure for arg2 according to char_arg1		

sym_abs(char_arg1, arg2)	absolute symmetry measure for arg2 according to char_arg1		
sym_rel(char_arg1, arg2)	relative symmetry measure for arg2 according to char_arg1		
ens(arg)	whole run ensemble	dim(res) = dim(arg)+1 ext(res,dim(res)) = number_of_runs coord(res,dim(res)) = name = run values = equidist_end 1(1) number_of_runs	

Tab. 8.14 *Syntax of the filter argument 1 for local sensitivity operators*

Placeholder	Explanation
<filter>	' { <select_operator$_1$> {, <select_operator$_2$> ... {, <select_operator$_3$> } ... } } '
<select_operator>	[self | seli | sels] { _<factor_val_type>} (<factor_val_range>) with self = select **f**actor range seli = select **i**ncrement range sels = select **s**ign range (only for sens_abs and sens_rel)
<factor_val_type>	specification how to interpret <val_range> i as a range of position **i**ndices (always count from 1) for self and seli s as a range of **s**ampled increment values for seli n as a range of factor **n**ames (sequ. as in <model>.edf) for self as a range of signs for sels
<factor_val_range>	[(<val$_1$> { : <val$_2$> }) | (*)] for <val$_2$> = <nil> : <val$_2$> = <val$_1$> (*) : use all values from <factor_name> <val$_i$> = <int_val$_i$> for <val_type> = i <val$_i$> = <real_val$_i$> for <val_type> = s <val$_i$> = <factor$_i$> for <val_type> = n (self) <val$_1$> = [+ | -] and <val$_2$> = <nil> for <val_type> = n (sels)

The following rules hold for the filter argument in local sensitivity operators:

- Generally, by the filter argument char_arg1 those runs from the run ensemble are selected (here interpreted as filtered) that are used for the formation of the result.
  Consequently, if no filter is specified all runs are used:
  `sens_abs(' ',atmo_g)`
  The filter operator has to be specified only if values are to be restricted by corresponding factor values, increment values and/or sign ranges.
- For the above three select operators self, seli and sels the factor value type is redundant if the factor value range represents the full range of values by [self | seli | sels] `(*)`:
  `self(*) = self_n(*) = self_i(*)` and all are redundant.
- Each select operator can be applied only once within the filter argument.
- For <val_type> = i, i.e. if a factor value range is specified by position indices those factors are selected for self and/or those increments are selected for seli that correspond to the specified position indices. Position indices are assigned from index 1 to the factors and or increments according to their specification sequence in the corresponding experiment description file <model>.edf.
- If more than one factor, increment value and/or sign was selected by the filter argument arg1 it contributes with an additional dimension to the result of the local sensitivity operator:
  - For factors an additional dimension factor_sequ
  - For increments an additional dimension incr

- For signs an additional dimension sign

is appended to the dimensions of the argument arg2 to form the result of the local sensitivity operator. The extent of this additional dimension corresponds to the defined and/or selected number of factors, increment values and/or signs. For a definition of the additional dimensions check Tab. 10.11.
Firstly, dimension factor_sequ is appended on demand, secondly dimension incr and thirdly dimension sign.

Having a model output variable definition as in Example 5.1 on page 31 and
assuming address_default=coordinate in <model>.cfg
Assume the experiment description file (4) from Example 6.4 on page 60
then in result-processing

```
sens_abs(' ',atmo_g)          absolute sensitivity measure for atmo_g
                              for all factors, increments and signs
                              Dimensionality = 4
                              Coordinates = time , factor_sequ , incr , sign
                              Extents = 20 , 3 , 4 , 2
sens_rel('sels_n(+),self_i(1)',atmo_g)
                              relative sensitivity measure for atmo_g
                              for factor p1 and all positive increments
                              Dimensionality = 2
                              Coordinates = time , incr
                              Extents = 20 , 4
sens_abs('seli_s(0.001:0.05)',atmo_g)
                              absolute sensitivity measure for atmo_g
                              for all factors, increment values 1 to 3 and all signs
                              Dimensionality = 4
                              Coordinates = time , factor_sequ , incr , sign
                              Extents = 20 , 3 , 3 , 2
lin_abs('seli_s(0.001:0.05)',atmo_g)
                              absolute linearity measure for atmo_g
                              for all factors and increment values 1 to 3
                              Dimensionality = 3
                              Coordinates = time , factor_sequ , incr , sign
                              Extents = 20 , 3 , 3
```

Example file: world.post_f

Example 8.8 *Experiment post-processing operators for local sensitivity analysis*

8.4.5 Monte Carlo Analysis

Tab. 8.15 shows experiment specific operators for Monte Carlo analysis that can be used in post-processing besides the general multi-run aggregation operators listed in Tab. 8.9 on page 100 and supplemented with a suffix _e.

Tab. 8.15 *Experiment specific operators for Monte Carlo analysis (without standard aggregation / moment operators)*

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 84)	Argument value restriction
same as in Tab. 8.9 with suffix _e	see Tab. 8.9	see Tab. 8.9	see Tab. 8.9
cnf(real_arg1, arg2)	positive distance of confidence measure from mean avg_e(arg2)	(1) real_arg1 probability of error	arg1 = [0.001 | 0.01 | 0.05 | 0.1]
cor(arg1, arg2)	correlation coefficient between arg1 and arg2	(2.1)	
cov(arg1, arg2)	covariance between arg1 and arg2	(2.1)	
ens(arg)	whole run ensemble	dim(res) = dim(arg)+1 ext(res,dim(res)) = number_of_runs coord(res,dim(res)) = name = run values = equidist_end 1(1) number_of_runs	
krt(arg)	kurtosis (4^{th} moment)	(1)	
med(arg)	median	(1)	
qnt(real_arg1, arg2)	quantile of arg2	(1) real_arg1 quantile value	0. ≤ arg1 ≤ 100.
reg(arg1, arg2)	linear regression coefficient to forecast arg2 from arg1	(2.1)	
rng(arg)	range = max_e(arg) - min_e(arg)	(1)	
skw(arg)	skewness (3^{rd} moment)	(1)	
stat_full(real_arg1, real_arg2, real_arg3, real_arg4, arg5)	full basic statistical measures of arg5	dim(res) = dim(arg)+1 ext(res,dim(res)) = 10 coord(res,dim(res)) = name = stat_measure values = equidist_end 1(1)10	arg1, arg2 = [0.001 | 0.01 | 0.05 | 0.1] arg1 < arg2 probability of error for confidence distance measure 0. ≤ arg3 < arg4 ≤ 100. quantile values

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 84)	Argument value restriction
stat_red(real_arg1, real_arg2, arg3)	reduced basic statistical measures of arg3	dim(res) = dim(arg)+1 ext(res,dim(res)) = 7 coord(res,dim(res)) = name = stat_measure values = equidist_end 1(1)7	arg1, arg2 = [0.001 | 0.01	0.05	0.1] arg1 < arg2 probability of error for confidence distance measure

The following explanations hold for the operators in Tab. 8.15:

- The operators **stat_full** and **stat_red** supply basic statistical measures for their last argument. Both operators are stand-alone operators: They must not be operands of any other operator. Contrary, their last argument can be composed from other non-multi-run operators. To store the statistical measures, dimensionality of both operators is that of their last argument, appended by an additional dimension with an extent of 10 and/or 7. Appended coordinate description is pre-defined by SimEnv (cf. Tab. 10.11).

  These ten data fields (for operator stat_full) and/or seven data fields (operator stat_red) correspond to the following statistical measures:
  1. Deterministic run (run number 0)
  2. Run ensemble minimum
  3. Run ensemble maximum
  4. Run ensemble mean
  5. Run ensemble variance
  6. Run ensemble positive distance of confidence measure from run ensemble mean for probability of error real_arg1
  7. Run ensemble positive distance of confidence measure from run ensemble mean for probability of error real_arg2

  Only for operator stat_full:
  8. Run ensemble median
  9. Run ensemble quantile for quantile value real_arg3
  10. Run ensemble quantile for quantile value real_arg4

  The operator stat_red was introduced because computation of the median and quantiles consumes a lot of auxiliary storage space. For the definition of the statistical measures check the corresponding single operators in Tab. 8.9 and Tab. 8.15. Both operators were designed for application of an appropriate visualization technique in result evaluation in future.

Having a model output variable definition as in Example 5.1 on page 31 and
assuming address_default=coordinate in world_*.cfg
Assume the Monte Carlo experiment from Example 6.5 (5) on page 63
then in experiment post-processing

`avg_e(p1*atmo(*,*,1,19:20))`	global run ensemble mean of p1*atmo for level 1 and the last two decades Dimensionality = 3 Coordinates = lat , lon , time Extents = 45 , 90 , 2
`avg(atmo(*,*,1,19:20))`	global mean of atmo for level 1 and the last two decades for run number 0 Dimensionality = 0 Coordinates = (without) Extents = (without)

```
ens(atmo(*,*,1,20)                    run ensemble values of atmo for level 1 and the last decade
                                      Dimensionality = 3
                                      Coordinates = lat , lon , run
                                      Extents = 45 , 90 , 250
minprop_e(atmo(*,*,1,19:20))          run ensemble run number for level 1 and the last two
                                      decades
                                      where the minimum of atmo is reached the first time
                                      Dimensionality = 3
                                      Coordinates = lat , lon , time
                                      Extents = 45 , 90 , 2
var_e(atmo(*,*,1,19:20))-atmo(*,*,1,19:20)
                                      anomaly for run ensemble variance from the default
                                      (nominal) run for level 1 the last two decades
                                      Dimensionality = 3
                                      Coordinates = lat , lon , time
                                      Extents = 45 , 90 , 2
var_e(atmo(*,*,1,19:20)-run('0',atmo(*,*,1,19:20)))
                                      global run ensemble variance of the anomaly of atmo for
                                      level 1 and the last two decades.
                                      Differs normally from the previous result expression
                                      Dimensionality 4
                                      Coordinates = lat , lon , time
                                      Extents = 45 , 90 , 4 , 20
hgr_e('bin_no',0,0.,0.,min_l('10',atmo(20:-20,*,1,20)))
                                      histogram with 25 bins for the zonal tropical minima
                                      for level 1 and the last decade. Bin bound extremes are
                                      derived from the values of the last argument of the operator
                                      hgr_e.
                                      Dimensionality = 2
                                      Coordinates = lat , bin_no
                                      Extents = 11 , 25
stat_full(0.01,0.05,25,75, min_l('10',atmo(20:-20,*,1,20)))
                                      full basic statistical measures for the zonal tropical minima
                                      of atmo for level 1 and the last decade
                                      Dimensionality = 2
                                      Coordinates = lat , stat_measure
                                      Extents = 11 , 10

                                                              Example file: world.post_e
```

Example 8.9 *Experiment post-processing operators for Monte Carlo analysis*

8.4.6 Optimization

The goal of an optimization experiment is to minimize a cost function by determining the corresponding optimal point in the factor space. Nevertheless, the specified model output from all single runs is stored during the experiment.

Tab. 8.16 *Experiment specific operator for the optimization experiment type*

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
same as for Monte Carlo analysis	see Section 8.4.5	see Section 8.4.5	see Section 8.4.5

While the single run that corresponds to the optimal cost function can be post-processed in the single-run modus, the whole experiment can be post-processed as a Monte Carlo analysis. Keep in mind that the factors do not follow a pre-defined distribution.

8.5 User-Defined and Composed Operators / Operator Interface

Besides application of built-in operators during experiment post-processing SimEnv enables construction and application of user-defined and composed post-processing operators. A user-defined operator is supplied by the user in the form of a stand-alone executable that is to perform the operator. Contrarily, a composed operator can be derived from both built-in and user-defined operators to generate more complex operators. User-defined and composed operators are announced to the environment in a user-defined operator description file <model>.odf by their names and the number of character, integer constant, real constant and "normal" arguments. This information is used to check user-defined and composed operators syntactically during experiment post-processing and by the SimEnv service simenv.chk. Sequence of the operator arguments types follows the same rule as for built-in operator (cf. Section 8.1.4).

A user-defined operator itself is a stand-alone executable that is executed during the check and the computation of the operator chain. While the main program of this executable is made available by SimEnv the user has to supply two functions in C/C++ or Fortran with pre-defined names that represent the check and the computational part. For declaration of both functions SimEnv comes with a set of operator interface functions. They can be used among others to get dimensionality, length, extents and coordinates of an argument and to get and check argument values and to put operator results.

For a composed operator the operator description file <model>.odf simply holds the definition of the corresponding operator chain composed from built-in and user-defined operators and using formal arguments.

8.5.1 Declaration of User-Defined Operator Dynamics

User-defined operators consist of a declarative and a computational part, that are described in one source file in two C/C++ or Fortran functions (cf. Tab. 8.17):

- Function simenv_check_user_def_operator
  This is the declarative part of the operator. The consistency of the non-character operands can be checked with respect to dimensionality, dimensions and coordinates as well as the values of character arguments can be checked. Dimensionality, extents and coordinates of the result have to be defined, normally in dependence on the argument information.
- Function simenv_compute_user_defined_operator
  This is the computational part of the operator. In the computational part the result of the operator in dependency of its operands is computed.

A function value ≠ 0 of simenv_check_user_def_operator( ) should be set according to the following rules:

- If appropriate, forward function value from the operator interface function simenv_chk_2args_[f | c] (see below) to the function value of simenv_check_user_def_operator( ). The corresponding error message is reported automatically by the experiment post-processor. Return code 4 from simenv_chk_2args_[f | c] is only an information and no warning and is not reported.
- Other detected inconsistencies between operands have to be reported to the user by a simple print-statement within simenv_check_user_def_operator. The corresponding return code has to be greather than 5.

Tab. 8.18 summarizes these SimEnv operator interface functions that can be applied in the declarative and computational part written in Fortran or C/C++ (postfix f for Fortran, c for C/C++) to get and put structure information. In this table the input and output data types are documented for functions used in Fortran. For C/C++ the corresponding data types are valid. Implementation of the functions for C/C++ is based on a call by reference for the function arguments.

Tab. 8.17 *Operator interface functions for the declarative and computational part*

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
Functions to host the declarative and computational part in usr_opr_<opr>.[f | c | cpp]			
simenv_ check_user_ def_operator ( )	check consistency of operator arguments and defines dimensionality and dimensions of result	integer*4 simenv_ check_user_ def_operator (function value)	return code = 0 ok ≠ 0 inconsistency between operands
simenv_ compute_user_ def_operator (res)	compute result of the operator in dependency on operands	real*4 res(1) (output)	result vector of the operator
		integer*4 simenv_ compute_user_ def_operator (function value)	return code = 0 ok ≠ 0 user-defined interrupt of calculation **Operator results of a dimensionality > 1 have to be stored to the field res using the Fortran storage model (cf. Section 15.7 - Glossary).**

Tab. 8.18 *Operator interface functions to get and put structural information*

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value Description
Functions to get and put structure information in the declarative and computational part			
simenv_ get_char_arg_ [f | c] (iarg, char)	get string and string length of a character argument	integer*4 iarg (input)	argument number
		character*(*) char (output)	string of the character argument Declare char with a suffficient length.
		integer*4 simenv_ get_char_arg_ [f | c] (function value)	length of character argument
simenv_ get_dim_arg_ [f | c] (iarg, iext)	Iarg > 0: get dimensionality and extents of an argument iarg = 0: get dimensionality and extents of the result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 iext(9) (output)	extents of argument / result iext(1) ... iext(simenv_get_dim_arg_[f | c]...)
		integer*4 simenv_ get_dim_arg_ [f | c] (function value)	dimensionality of argument / result

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value Description
simenv_ get_len_arg_ [f | c] (iarg)	iarg > 0: get length of an argument iarg = 0: get length of the result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 simenv_ get_len_arg_f (function value)	length of argument / result
simenv_ get_nr_arg_ [f | c] ( )	get number of arguments of the current operator	integer*4 simenv_ get_nr_arg_ [f | c] (function value)	number of arguments
simenv_ get_type_arg_ [f | c] (iarg)	iarg > 0: get data type of an argument iarg = 0: get data type of the result	integer*4 iarg (input)	argument number, 0 for result
		integer*4 simenv_ get_type_arg_f (function value)	type of argument / result = -1 byte = 4 float = -2 short = 8 double = -4 int
simenv_ get_co_chk_ modus_ [f | c] ( )	get level of coordinate check for arguments according to <model>.cfg	integer*4 simenv_ get_co_chk_ modus_ [f | c] (function value)	level of coordinate check for arguments = 0 without = 1 weak = 2 strong
simenv_ get_co_arg_ [f | c] (iarg, ico_nr, ico_beg_pos, co_name)	get formal coordinate numbers and formal coordinate begin value positions of an argument	integer*4 iarg (input)	argument number
		integer*4 ico_nr(9) (output)	formal numbers of the coordinates ico_nr(1) ... ico_nr(simenv_get_dim_ arg_[f | c]...)
		integer*4 ico_beg_pos(9) (output)	formal begin value positions of the coordinates ico_beg_pos(1) ... ico_beg_pos(simenv_get_dim_ arg_[f | c]...)
		character*20 co_name(9) (output)	coordinate names co_name(1) ... co_name(simenv_get_dim_ arg_[f | c]...)
		integer*4 simenv_ get_co_arg_ [f | c] (function value)	return code = 0 ok
simenv_ get_co_val_ [f | c] (ico_nr, ico_pos, co_val)	get for a coordinate a coordinate value at a specified position	integer*4 ico_nr (input)	formal number of the coordinate (from simenv_get_co_arg_[f | c])
		integer*4 ico_pos (input)	formal position within all coordinate values of the value to get. The smallest ico_pos to use corresponds to the value ico_beg_pos from the function simenv_get_co_arg_[f | c]

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value Description
	Application of this function in simenv_check_ user_def_operator for coordinate bin_mid results in an error	real*4 co_val (output)	coordinate value For non-monotonic coordinate values: Do not get value number ico_pos but the (ico_pos) th smallest value (sort values in increasing manner)
		integer*4 simenv_ get_co_arg_ [f | c] (function value)	return code = 0 ok = 1 ico_pos out of range = 2 storage exceeded
simenv_ chk_2args_ [f | c] (iarg1, iarg2)	check two arguments on same dimensionality, extents and coordinates **If appropriate forward return code ≠ 0 to the function value of simenv_check_ user_def_ operator( )**	integer*4 iarg1 (input)	argument number
		integer*4 iarg2 (input)	argument number
		integer*4 simenv_ chk_2args_ [f | c] (function value)	return code = 0 ok = 1 differing dimensionalities = 2 differing extents = 3 differing coordinates according to the sub-keyword 'coord_check' in <model>.cfg = 4 iarg1 = iarg2
simenv_ put_struct_res_ [f | c] (inplace, idimens {, iext, ico_nr, ico_beg_pos })	put - potential in-place-storage - dimensionality - extents - formal coordinate number - formal coordinate value begin number of the result Currently, only coordinates from the arguments can be assigned to the result. **Must be applied in the declarative part and only there.**	integer*4 inplace (input)	potential inplace-indicator for result. result can be computed in-place with the following non-character arguments = -1 all = 0 none > 0 e.g. = 135 with arguments 1, 3 and 5
		integer*4 idimens (input)	dimensionality of the result
		integer*4 iext(9) (input)	only for idimens > 0: extents of the result iext(1) ... iext(idimens)
		integer*4 ico_nr(9) (input)	only for idimens > 0: formal coordinate numbers of the result ico_nr(1) ... ico_nr(idimens)
		integer*4 ico_beg_pos(9) (input)	only for idimens > 0: formal coordinate begin position for formal coordinate number ico_nr of the result ico_beg_pos(1) ... ico_beg_pos(idimens)
		integer*4 simenv_ put_dim_res_ [f | c] (function value)	return code = 0 ok ≠ 0 inconsistency between operands

All of these operator interface functions return -999 as an error indicator if an argument iarg is invalid. Output arguments are set to -999 as well.

Tab. 8.19 summarizes these SimEnv operator interface functions that can be applied in the computational part written in Fortran or C/C++ (postfix f for Fortran, c for C/C++) to get and check argument values and put results. In this table the input and output data types are documented for functions used in Fortran. For C/C++ the corresponding data types are valid. Implementation of the functions for C/C++ is based on a call by reference for the function arguments.
To handle real*4 underflow and overflow during computation of the operator results with real*4 argument values it is advisible to compute operator results temporarily as real*8 values and afterwards to transform these values back to the final real*4 operator result by the function simenv_clip_undef_[f | c].

Tab. 8.19 *Operator interface functions to get / check / put arguments and results*

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value Description
Functions to get and check argument values and to put results in the computational part			
simenv_ get_arg_ [f | c] (iarg,index)	get value of a non-character argument with index index	integer*4 iarg (input)	argument number
		integer*4 index (input)	vector index of an argument
		real*4 simenv_ get_arg_ [f | c] (function value)	value of argument iarg at index index **Operands of any type are transferred by simenv_get_arg_[f | c] to a real*4 / float representation.** **Operands of a dimensionality > 1 are forwarded to user-defined operators as one-dimensional vectors, using the Fortran storage model (cf. Section 15.7 - Glossary). Adjust the second argument of simenv_get_arg_[f | c] (index) accordingly.**
simenv_ clip_undef_ [f | c] (value)	**overflow:** set a real*8 value to an undefined real*4 result if appropriate **underflow:** set a real*8 value to real*4 0. if appropriate	real*8 value (input)	value to be checked

		real*4
		simenv_
		clip_undef_
		[f | c]
		(function value)
			Example:
			`res(i)=simenv_clip_undef_[ f | c ]`
			`(value)`

simenv_ chk_undef_ [f | c] (value)	check whether value is undefined before processing it	real*4 value (input)	argument value to be checked
		integer*4 simenv_ is_undef_ [f | c] (function value)	= 0 value is defined = 1 value is undefined
simenv_ put_undef_ [f | c] ( )	set a result value as undefined	real*4 simenv_ put_undef_ [f | c] (function value)	Example: `res(i)=simenv_put_undef_[ f | c ] ( )`

- In SimEnv the declarative and computational part of a user-defined operator <opr> is hosted in a source file usr_opr_<opr>.[f | c | cpp]. The assigned executable has the name usr_opr_<opr> and has to be located in that directory that is stated in <model>.cfg as the hosting directory opr_directory for user-defined operators.
- The include file simenv_opr_f.inc and simenv_opr_c.inc from the inc sunbirectory of the SimEnv home directory can be used in user-defined operators to declare the SimEnv operator interface functions for Fortran and/or C/C++ (cf. also Tab. 10.6).
- Apply the shell script
  simenv_opr_[f | c | cpp].lnk <opr>
  from the SimEnv library directory $SE_HOME/lib to compile and link from usr_opr_<opr>.[f | c | cpp] an executable usr_opr_<opr> that represents the user-defined operator <opr>. Like the main program for the operator also the object $SE_HOME/bin/simenv_opr.o is supplied by SimEnv. This object file has to be linked with usr_opr_<opr>.o and the object library $SE_HOME/lib/libsimenv.a.
- Tab. 15.13 lists the additionally used symbols when linking a user-defined operator.
- In Section 15.3 on page 184 implementation of the user-defined operator matmul_[f | c] is described in detail. It corresponds to the built-in operator matmul. Additionally, check the user-defined operators from Tab. 15.6 and apply them during experiment post-processing.

8.5.2 Undefined Results in User-Defined Operators

Check always by the SimEnv operator interface function simenv_chk_undef(val) (cf. Tab. 8.19) whether an argument value val is undefined before it is processed.

Set a result to be undefined by the SimEnv operator interface function simenv_put_undef( ) (cf. Tab. 8.19)
Check usr_opr_matmul_[f | c].[f | c] in Section 15.3 or usr_opr_div.f in the example directory $SE_HOME/exa of SimEnv for more detailed examples.

If things go so wrong that computation of the whole result expression has to be stopped it is possible to alternatively
- Set all elements of the results to be undefined
- Set simenv_compute_user_def_operator ≠ 0 (otherwise set it always = 0)
- In both cases application of the following operators in the operator chain of the result expression will be suppressed and consequently computation of the result expression will be stopped
- Check usr_opr_char_test.f for a detailed example

8.5.3 Composed Operators

A composed operator is an operator chain composed from built-in and user-defined operators. The concept of composed operators enables construction of more complex operators from built-in and user-defined ones. A composed operator is defined with formal arguments that are used in the operator chain as arguments. Formal arguments are replaced by current arguments when applying a composed operator during experiment post-processing. In this sense, the definition of a composed operator in SimEnv corresponds to the definition of a function in a programming language: When calling the function formal arguments are replaced by current arguments. Consequently, composed operators offer the same flexibility as built-in or user-defined operators.

Like built-in and user-defined operators, a composed operator can have nine formal arguments at maximum. Sequence of these arguments is also the same as for the other operators: Character arguments followed by integer constant arguments, real constant arguments and normal arguments.

For composed operators the operand set (cf. Section 8.1.2) to form the operator by a chain of operators is restricted to
- Constants in integer and real / float notation
- Character strings
- Operator results from built-in and user-defined operators

Not allowed as operands are
- Model output variables
- Experiment factors

- Composed operators
- Macros

Additionally have to be used

- Formal arguments arg1 ,…, arg9

Check the following example how to specify composed operators.

composed operator name	character argument	"normal" argument	composed operator definition
rel_count (	arg1 ,	arg2) =	`100 * count(arg1,arg2) / count('all',arg2)`
error_1 (	arg1 ,	arg2) =	`count(arg1,arg2) * hgr(arg1,0,0.,0.,arg2)`
error_2 (		arg1) =	`arg1 * hgr('bin_mid',10,0.,0.,arg1)`

Having a model output variable definition as in Example 5.1 on page 31
then for example, the operator rel_count can be applied by

```
rel_count('def',bios)
rel_count('def',bios(c=20:-20,*,1))
rel_count('undef',100*bios)
```

Example 8.10 *Composed operators*

Composed operators are checked syntactically by the SimEnv service simenv.chk. When performing simenv.chk validity of the following information is **not** cross-checked between formal arguments:

- Character arguments of operators
  Example: The composed operator error_1 is considered by simenv.chk to be valid though
  argument 1 of operator count is limited to values ['all' | 'def' | 'undef'] and
  argument 1 of operator hgr is limited to values ['bin_no' | 'bin_mid']
- Use of "normal" formal arguments in the operator chain with respect to their dimensionality, extents and coordinates
  Example: The composed operator error_2 is considered by simenv.chk to be valid though the dimensionality of the operator hgr in this constellation is always higher than that of the argument arg1 and consequently, multiplication between arg1 and hgr(.) is impossible.

8.5.4 Operator Description File <model>.odf

<model>.odf is an ASCII file that follows the coding rules in Section 11.1 on page 141 with the keywords, names, sub-keywords, and values as in Tab. 8.20. <model>.odf announces the user-defined and composed operators by their names, and the number of character, integer constant, real constant, and normal arguments that belong to an operator. Additionally, <model>.odf hosts for composed operators the corresponding operator chain using formal arguments. <model>.odf is expoited to check a user-defined and/or composed operator syntactically when performing it during experiment post-processing.

Tab. 8.20 *Elements of an operator description file <model>.odf*

keyword	name	sub-keyword	Line type	Max. line nmb.	values	Explanation
general	<nil>	descr	o	any	<string>	general operator descriptions
opr_defined	<user_defined_operator_name>	descr	o	1	<string>	operator description
		arguments	m	1	<int_val$_1$>, <int_val$_2$>, <int_val$_3$>, <int_val$_4$>	number of arguments defined for the operator: <int_val$_1$> ≥ 0: character arguments <int_val$_2$> ≥ 0: integer constant arguments <int_val$_3$> ≥ 0: real constant arguments <int_val$_4$> > 0: “normal” arguments
opr_composed	<composed_operator_name>	descr	o	1	<string>	operator description
		arguments	m	1	<int_val$_1$>, <int_val$_2$>, <int_val$_3$>, <int_val$_4$>	number of arguments defined for the operator. Explanations and restrictions are the same as for a user-defined operator
		define	m	≥ 1	<string>	operator definition string Operator definition can be arranged at a series of define-lines in analogy to the rules for result expressions (cf. Section 8.1.1).

To Tab. 8.20 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- The sequence of the four integer values <int_val$_1$> ,…, <int_val$_4$> follows the sequence of arguments in built-in, user-defined and composed operators.
- The sum <int_val$_1$> +…+ <int_val$_4$> has to be less equal 9.
- Use the SimEnv service simenv.chk to check user-defined and composed operators.

```
general                    descr       Operator description for the
general                    descr       examples in the SimEnv User Guide

opr_defined   matmul_f     descr       matrix multiplication (in Fortran)
opr_defined   matmul_f     arguments   0,0,0,2

opr_defined   matmul_c     descr       matrix multiplication (in C)
opr_defined   matmul_c     arguments   0,0,0,2

opr_defined   corr_coeff   descr       correlation coefficient r
opr_defined   corr_coeff   arguments   0,0,0,2

opr_defined   div          descr       arithmetic division
opr_defined   div          arguments   0,0,0,2

opr_defined   simple_div   descr       division without undefined-check
opr_defined   simple_div   arguments   0,0,0,2
```

```
opr_defined   char_test    descr       test character arguments
opr_defined   char_test    arguments   2,0,0,1

opr_composed  rel_count    descr       relative count [%]
opr_composed  rel_count    arguments   1,0,0,1
opr_composed  rel_count    define      100*count(arg1,arg2)/
opr_composed  rel_count    define      count('all',arg2)
```

Example files: world_[f | c | cpp | py | ja | m | sh].odf

Example 8.11 *Operator description file <model>.odf*

8.6 Undefined Results

By performing operator chains and due to possibly unwritten model output during simulation parts of the intermediate and/or final result values can be undefined within the float data representation.

If an operand is completely undefined the computation of the result is stopped without evaluating the following operands and operators.

For undefined / nodata value representation check Tab. 10.13.

8.7 Macros and Macro Definition File <model>.mac

- In experiment post-processing a macro is an abbreviation for a result expression, consisting of an operator chain applied on operands.
- Generally, they are model related and they are defined by the user.
- Macros are identified in experiment post-processing expressions by the suffix _m.
- A macro is plugged into a result expression by putting it into parentheses during parsing:
  Example: `equ_100yrs_m*test_mac_m`
  from Example 8.12 below is identical to
  `(avg(atmo(c=20:-20,*,c=1,c=11:20))-400)*(1+(2+3)*4)`
- Macros must not contain macros.
- Use simenv.chk to check macros. During the macro check validity of the following information is not checked:
  - Un-pre-defined character arguments of built-in operators (cf. Tab. 15.10)
  - Integer or real constant arguments of built-in operators (cf. Tab. 15.11)
  - Character arguments of user-defined operators
  - Operators with respect to dimensionality and dimensions of its operands

In SimEnv macros are defined in the file <model>.mac. <model>.mac is an ASCII file that follows the coding rules in Section 11.1 on page 141 with the keywords, names, sub-keywords, and values as in Tab. 8.21. <model>.mac describes the user-defined macros.

To Tab. 8.21 the following additional rules and explanations apply:
- For the description of **line type** check Tab. 11.4 on page 143.
- Values for sub-keywords ‘descr’ and ‘unit’ are not evaluated during parsing a result expression.

Tab. 8.21 *Elements of a macro description file <model>.mac*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	general macro descriptions
macro	<macro_ name>	descr	o	1	<string>	macro description
		unit	m	1	<string>	unit of the value of the macro
		define	m	≥ 1	<string>	macro definition string macro definition can be arranged at a series of define-lines in analogy to the rules for result expressions (cf. Section 8.1.1).

```
general                 descr       Macro definitions for the
general                 descr       examples in the SimEnv User Guide

macro  equ_100yrs       descr       2nd century tropical level 1 average
macro  equ_100yrs       unit        without
macro  equ_100yrs       define      avg(atmo(c=20:-20,*,c=1,c=11:20))

macro  tst              descr       test macro
macro  tst              define      1+(2+3)*
macro  tst              define      4
```

Example files: world_[f | c | cpp | py | ja | m | sh].mac

Example 8.12 *User-defined macro definition file <model>.mac*

8.8 Wildcard Operands &v& and &f&

In SimEnv, wildcard operands offer a convenient approach to compute a result expression successively for all defined model output variables and experiment factors. Wildcard operands are used in the same manner as normal operands when defining a result expression. There are two wildcard operands at disposal:

&v& wildcard operand for any model output **v**ariable
&f& wildcard operand for any experiment **f**actor

When applying in a result expression only one wildcard type (i.e., either &v& or &f&) the result expression is performed repetitively where the wildcard is replaced successively by all model output variables and experiment factors, respectively. When applying both &v& and &f& in a result expression the result expression is performed for the Cartesian product of all model output variables and experiment factors.

Wildcard operands must not be used in macro definitions (cf. Section 8.7). The wildcard operand &v& for model output variables cannot be restricted to a portion of the variable by appending a sub-specification in brackets as explained in Section 8.1.3 (e.g., &v&(i=3:10) is not allowed).

Note that the strings &v& and &f& are only substituted in the result string by model variables and/or model factors if they are

- prefixed by [(| + | - | / | * | begin of result string] and
- postfixed by [(| + | - | / | * | end of result string]

Having a model output variable definition as in Example 5.1 on page 31 and assuming the experiment description file (3b) from Example 6.3 on page 58 then in result-processing

```
behav(' ',sin(&v&))          results in
                             behav(' ',sin(atmo))
                             behav(' ',sin(bios))
                             behav(' ',sin(atmo_g))
                             behav(' ',sin(bios_g))
behav(' ',&v&*&f&)           results in
                             behav(' ',atmo*p1)
                             behav(' ',bios*p1)
                             behav(' ',atmo_g*p1)
                             behav(' ',bios_g*p1)
                             behav(' ',atmo*p2)
                             behav(' ',bios*p2)
                             behav(' ',atmo_g*p2)
                             behav(' ',bios_g*p2)
```

Example 8.13 *Experiment post-processing with wildcard operands*

8.9 Saving Results

The result files <model>.res<simenv_res_char>.[nc | ieee | ascii] and <model>.inf<simenv_res_char>.[ieee | ascii] contain all the model and experiment information for further processing of results.

9 Visual Experiment Evaluation

Experiment evaluation in SimEnv is based on its visualization framework SimEnvVis. The SimEnvVis approach is to apply visualization techniques to the output data, derived during experiment post-processing and stored in NetCDF format. SimEnvVis does not belong to the standard SimEnv distribution. It can be obtained from the SimEnv developers on request.

Analysis and evaluation of post-processed data selected and derived from large amount of relevant model output benefits from visualization techniques. Based on metadata information of the post-processed experiment type, the applied operator chain, and the dimensionalities of the post-processor output pre-formed visualization modules are evaluated by a suitability coefficient how they can map the data in an appropriate manner.
The visualization modules offer a high degree of user support and interactivity to cope with multi-dimensional data structures. They cover among others standard techniques such as isolines, isosurfaces, direct volume rendering and a 3D difference visualization techniques (for spatial and temporal data visualization). These techniques are accompanied by parallel corrdinates, graphical table and scatterplot matrixes techniques. Furthermore, approaches to navigate intuitively through large multi-dimensional data sets have been applied, including details on demand, interactive filtering and animation.
Using the OpenDX platform, OpenGL and Ferret visualization techniques have been designed and implemented, suited in the context of analysis and evaluation of derived multi-run output functions.

Currently, visual experiment evaluation is the only SimEnv service that comes with a graphical user interface. In this user interface a help-services is implemented that should be used to gather additional information on how to select post-processed results for visualization and on visualization techniques provided by SimEnvVis. Additionally, a SimEnvVis user documentation is available from the SimEnv website.

Visualization of post-processed experiment output is started by the SimEnv service simenv.vis (check Section 10.2) and directly during experiment post-processing by the service simenv.res if in the file <model>.cfg (check Section 10.1) this feature is enabled by

```
postproc        visualization        yes
```

At PIK, the SimEnvVis framework is installed at viss01.pik-potsdam.de. Access to viss01 is requested by the SimEnv service simenv.key. Check Section 10.2 for more information.

To apply SimEnvVis, an X11 server must run on the client's machine. On Windows systems this may be Hummingbird or Cygwin/X, on Mac machines an XTerm.

10 General Control, Services, User Files, and Settings

In the control file simenv_settings.txt general SimEnv settings are defined, while <model>.cfg is a model and workspace-related general configuration file to control preparation, performance and analysis of an experiment. Besides simulation performance and experiment post-processing SimEnv supplies a set of auxiliary services to check status of the model, to dump model and post-processor output and files and to clean a model from output files. General built-in settings reflect case sensitivity, nodata values and other information related to SimEnv.

10.1 General Configuration Files simenv_settings.txt and <model>.cfg

$SE_HOME/bin/simenv_settings.txt is the general SimEnv settings file. It is a case-sensitive ASCII file with the structure

<keyword> <sep> <value>

Tab. 10.1 lists the keywords and their values. Unless marked by (*), each of the keywords has to be used exactly one time, even it is not necessary for the current SimEnv installation. Keywords marked by (*) can be multiple specified.

Tab. 10.1 *Elements of the file simenv_settings.txt*

Keyword		Value
SimEnv_admin	(*)	email address of the SimEnv administrator
logfile_directory		directory to store SimEnv log files
SimEnv_home_directory2log	(*)	SE_HOME directory to store log files from
jms_login_node	(*)	login node for a compute cluster to access the job management system JMS
postproc_test_suite		specification of test mode for experiment post-processing
SimEnvVis_server_hostname		name of the visualization server that hosts the visualization component SimEnvVis
SimEnvVis_server_account		account on the visualization server for accessing SimEnvVis
SimEnvVis_home_directory		SimEnvVis home directory
SimEnvVis_working_directory		SimEnvVis working directory on the visualization server
server_SimEnv_home_directory		SE_HOME directory on the visualization server
ssh_local		ssh implementation of the client computer
scp_local		scp implementation of the client computer
ssh-keygen_local		ssh-keygen implementation of the client computer

In the ASCII file <model>.cfg general SimEnv control variables can be declared. <model>.cfg is workspace and model related and is an ASCII file that follows the coding rules in Section 11.1 on page 141 with the keywords, names, sub-keywords, and info as in Tab. 10.2.

Tab. 10.2 *Elements of a general configuration file <model>.cfg*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	general configuration description
		message_level	o	1	[info | warning | error]	specifies which message types to show
model	<nil>	out_directory	o	1	<directory>	model output directory
		out_format	o	1	[netcdf | ieee]	model output format
		out_separation	o	1	[yes | no]	indicates whether to store model output in a single file per single run or in one file per experiment
		slices	o	1	[no_f_c | py_ja_m]	indicates whether simenv_slice_* is not used / used for Fortran / C or is used for Python / Java / Matlab
		auto_interface	o	1	[no | all | f | c | py | sh]	indicates to generate include source code files for the semi-automated model interface for the corresponding languages
		structure	o	1	[standard | distributed | parallel]	indicates model structure with respect to experiment per-formance
experiment	<nil>	restart_ini	o	1	[no | yes]	perform <model>.ini for ex-periment re-start
		begin_run	o	1	<int_val>	begin single run number
		end_run	o	1	[last | <int_val>]	end single run number
		include_runs	o	1	<val_list>	single run numbers to include in the experiment
		exclude_runs	o	1	<val_list>	single run numbers to exclude from the experiment
		email	o	1	<string>	email notification address
postproc	<nil>	out_directory	o	1	<directory>	experiment post-processing output directory
		out_format	o	1	[netcdf | ieee | ascii]	experiment post-processing output format
		address_default	o	1	[coordinate | index]	experiment post-processing address default for model output variables
		coord_check	o	1	[strong | weak | without]	post-processing coordinate check by operators
		opr_directory	o	1	<directory>	directory the post-processors expects user-defined operator executables
		factors_in_output	o	1	[yes | no]	determine whether factor val-ues are stored in SimEnv model output
		visualization	o	1	[yes | no]	determine whether to directly visualize an entered result during experiment post-processing

To Tab. 10.2 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- **For keyword 'general', sub-keyword 'message_level'**:
  Message output is controlled by this information.
  Specify info to output errors and warnings and additional information
  warning to output errors and warnings
  error to output errors
  during any SimEnv service.
- **For keyword 'model', sub-keyword 'out_separation'**:
  Specify here whether SimEnv model output data for the whole run ensemble is stored into one file <model>.outall.[nc | ieee] or in single output files <model>.out<simenv_run_char>.[nc | ieee].
- **For keyword 'model', sub-keyword 'slices'**:
  If the model interface function simenv_slice_* are not applied or applied in Fortran or C/C++ models set the values to no_f_c. For applying them in a Python, Java or Matlab model set the value to py_ja_m. If in the overall model slices are used in Python, Java or Matlab and as well as in Fortran or C/C++ set the value to py_ja_m. Running a model with slices = py_ja_m results in a significant increase of CPU time consumption per call of simenv_slice_* and simenv_put_* since slice information is then stored in external files rather than internally as for slices = no_f_c.
- **For keyword 'model', sub-keyword 'auto_interface'**:
  Check Section 5.10.
- **For keyword 'model', sub-keyword 'structure'**:
  Check Section 5.11.
- **For keyword 'experiment', sub-keyword ['begin_run' | 'end_run' | include-runs | exclude_runs]:**
  With the exception of an optimization experiment and a Monte Carlo analysis with a stopping function SimEnv enables to perform an experiment partially by performing only a subset of all defined single simulation runs from the whole run ensemble (cf. Section 7.5 on page 74). To declare runs for including into a SimEnv experiment use either sub-keywords 'begin_run' and 'end_run' or sub-keyword 'include_runs'. For 'begin_run' and 'end_run' assign appropriate run numbers. Make sure that begin and end run represent integer run number (including run number 0) and that begin run ≤ end run. The value string "last" for 'end_run' always represents the last simulation run of the whole run ensemble. Alternatively, a list of integer run number values can be defined by using a value list for the sub-keyword 'include_runs'. The include set as defined by the sub-keywords 'begin_run' / 'end_run' / 'include_runs' can be reduced by specification of a list of integer run number values defined by the sub-keyword 'exclude_runs' using again a value list. Specification of 'exclude_runs' demands an explicit specification of either 'begin_run' and 'end_run' or of 'include_runs'. As a result, the experiment is performed on the intersection of (i) the number of runs as specified by the experiment definition in <model>.edf, (ii) the include set, and (iii) the exclude set.
- **For keyword 'experiment', sub-keyword 'email'**:
  After performing an experiment an email is sent to the email address specified in <string>. Specify always a complete address.
- **For keyword 'postproc', sub-keyword 'address_default'**:
  During experiment post-processing portions of multi-dimensional model output variables can be addressed by coordinate (c= ...) or index (i= ...) reference. A default is established here.
- **For keyword 'postproc', sub-keyword 'coord_check'**:
  During experiment post-processing feasibility of application of an operator on its operands is checked with respect to the coordinate description of the operands. Different levels of this check are possible. A default is established here.
- **For keyword 'postproc', sub-keyword 'factors_in_output'**:
  Special model interface constellations may lead to a situation that all factor values are not stored in SimEnv model output. This could happen when simenv_get_* was not used but another technique for getting factor values within the model. If specifying factors_in_output as 'no' adjusted factor values are derived from <model>.smp and <model>.edf.
- **For keyword 'postproc', sub-keyword 'visualization'**:
  Specifies whether to directly visualize an entered result during experiment post-processing.

Keep in mind to ensure consistency of control settings in <model>.cfg across different SimEnv services. As an example one has to run experimentation, experiment post-processing and dump with the same value for out_separation in <model>.cfg.

Tab. 10.3 lists the default values in the general configuration file in the case of absence of the appropriate sub-keyword.

Tab. 10.3 *Default values for the general configuration file*
***(*)**: in the case of absence of the appropriate sub-keyword*

keyword	sub-keyword	Default value (*)	For more information see
general	descr	<nil>	above
	message_level	info	above
model	out_directory	./	above
	out_format	NetCDF	Chapter 12
	out_separation	yes	above
	slices	no_f_c	above
	auto_interface	no	Section 5.10
	structure	standard	Section 5.11 and above
experiment	restart_ini	no	Section 7.4
	begin_run	0	Section 7.1 - 7.5
	end_run	last	Section 7.1 - 7.5
	include_runs	<nil>	above
	exclude_runs	<nil>	above
	email	<nil>	Section 7.1
postproc	out_directory	./	above
	out_format	NetCDF	Chapter 12
	address_default	coordinate	Section 8.1.3 and above
	coord_check	strong	Section 8.1.5 and above
	opr_directory	./	Section 8.5
	factors in output	yes	above
	visualization	yes	above

```
general       descr                 General configuration file for the
general       descr                 examples in the SimEnv User Guide
general       message_level         info

model         out_directory         mod_out
model         out_format            netcdf
model         out_separation        yes
model         auto_interface        f
model         structure             standard

experiment    begin_run             45
experiment    end_run               300
experiment    exclude_runs          file runs2exclude.dat

postproc      out_directory         res_out
postproc      out_format            netcdf
postproc      address_default       index
postproc      coord_check           strong
postproc      opr_directory         ./
postproc      visualization         no
```

Example 10.1 *User-defined general configuration file <model>.cfg*

10.2 Main and Auxiliary Services

The following SimEnv service commands are available from the sub-directory bin of the SimEnv home directory $SE_HOME. Besides experiment performance, experiment post-processing and visualization there are additional auxiliary SimEnv services to check input information consistency, to monitor the status of a running simulation experiment, to dump files of model and experiment post-processor output, to monitor SimEnv log files and to wrap up a SimEnv workspace.

Tab. 10.4 *SimEnv services*

SimEnv service	Use to
Main Services	
simenv.res <model> { [new | append | replace] } {<simenv_run_int>}	perform experiment **result** post-processing for run number <simenv_run_int> or for the whole run ensemble (<simenv_run_int> = -1, default). Before entering experiment post-processing those output files <model>.res<simenv_res_char>.[nc | ieee | ascii] and <model>.inf<simenv_res_char>.[ieee | ascii] with the highest two-digit number <simenv_res_char> are identified and new result files for <res+1> are written / the results are appended / or the result files are replaced by new ones.
simenv.rst <model>	**restart** an experiment (cf. Section 7.4)
simenv.run <model>	prepare and **run** an experiment (cf. Section 7.1)
simenv.vis <model> { [latest | <simenv_res_int>] }	perform **visual** post-processor output visualization with SimEnvVis for that NetCDF post-processor output file with the highest two digit number <simenv_res_char> (<simenv_res_char> = latest, default) or with the file number <simenv_res_char>. At PIK, visualization runs on a remote server.
Auxiliary Services	
simenv.chk <model>	**check** on model script files (<model>.run, <model>.rst, <model>.ini, <model>.end) check <model>.cfg <model>.edf <model>.odf <model>.gdf <model>.mdf <model>.mac existing model and post-processor output files generate pre-experiment output statistics
simenv.cln <model>	**clean** up model and experiment post-processor output files Deletes all model output files, post-processor output files, log-files, and auxiliary files of a model according to the settings in <model>.cfg
simenv.cpl <model> { <simenv_run_int> } { <file> }	**complete** sequence of SimEnv services simenv.chk, simenv.run, simenv.res, simenv.vis, simenv.dmp simenv.res is performed with input file <file> (if available) and interactively, for both optionally only for single run <simenv_run_int>.
simenv.cpy <model>	**copy** all SimEnv example files <model>* from the example directory $SE_HOME/exa to the current directory. Additionally, example files of user-defined operators and for models world_[f | c | cpp | py | ja | m | sh]* common user defined files are copied. All files are only copied if they do not already exist in the current workspace.
simenv.dmp <model> <dmp_modus>	**dump** SimEnv model output or experiment post-processor output files Files to dump have to match the SimEnv file name convention for model and/or post-processor output and are expected to be in the directories as stated in <model>.cfg. Model output variables and post-processor results in IEEE and/or ASCII format with a dimensionality greater than 1 are listed according to Fortran storage model for multi-dimensional fields (cf. Section 15.7 - Glossary).

SimEnv service	Use to
simenv.hlp <topics>	acquire basic SimEnv **help** information for the specified topics
simenv.key <user_name>	get password-free access to the SimEnv visualization server. Only for systems where the SimEnvVis visualization server is not hosted on the local machine – check $SE_HOME/bin/simenv_settings.txt Start this service only one time before the first access to simenv.vis and/or simenv.res or if the access does not work properly. An email will be sent from SimEnv when the access is enabled.
simenv.sta <user_name> {<begin_date>} {<end_date>} {<sort>}	generate log file **statistics** All SimEnv services are logged during their performance into log-files. The log-file directory is specified in $SE_HOME/bin/simenv_settings.txt. simenv.sta evaluates these log-files statistically and generates a report w.r.t SimEnv accesses, experiments, experiment post-processing and visualization.
simenv.sts <model> { <repetition_time> }	get the current **status** of an active simulation experiment. Start this service from the workspace the active simulation experiment was started from. This is the only service that can be started from a workspace where another service is active.
simenv.srv <user_name>	**serve** a request for password-free access to the SimEnv visualization server – only for systems where the SimEnvVis visualization server is not hosted on the local machine and only for the SimEnv administrator – check $SE_HOME/bin/simenv_settings.txt

- All but services simenv.cpy, simenv.hlp, simenv.key, simenv.sta and simenv.srv:
  Start a service only from the current workspace.
- All but service simenv.sts:
  A SimEnv service cannot be started from a workspace where an other SimEnv service is active.

10.3 Model Interface Scripts, Include Files, Link Scripts

Tab. 10.5 lists all these dot scripts and shell scripts that can / must be used in <model>.[ini | run | end].

Tab. 10.5 *Shell scripts and dot scripts that can be used in <model>.[ini | run | end]*
For built-in shell script variables in <model>.run see Tab. 10.10
***(*)**: this is not a dot script but a normal shell script with two arguments*

Dot script	Use status	Used for	See Section
<model>.ini			
simenv_ini_gams	mandatory	experiment init for GAMS models	5.7
simenv_ini_ja	mandatory	experiment init for Java models	5.5
simenv_ini_m	mandatory	experiment init for Matlab models	5.5
simenv_ini_py	mandatory	experiment init for Python models	5.5
<model>.run			
simenv_ini_sh	mandatory	init for any model	5.8
simenv_get_sh	optional	get a factor value as script variable	5.8
simenv_get_as	optional	get all factor names and adj. values to an ASCII file	5.9
simenv_run_gams	mandatory	run a GAMS model	5.7
simenv_run_mathematica	mandatory	run a Mathematica model	5.6
simenv_put_as (*)	optional	put ASCII file to SimEnv model output	5.9
simenv_put_as_simple (*)	optional	put ASCII file to SimEnv model output (simple mode)	5.9
simenv_end_sh	mandatory	end for any model	5.8
simenv_kill_process (*)	optional	kill a program / model after reaching a CPU time limit	7.2
<model>_[sh | as].inc	optional	semi-automated model interface at shell script /	5.10

Dot script	Use status	Used for	See Section
		ASCII level (cf. also Tab. 10.6)	
<model>.end			
simenv_end_gams	mandatory	experiment end for GAMS models	5.7

In Tab. 10.6 all that include files and link scripts are compiled that are provided by the simulation environment or generated by the user and/or automatically during performing a SimEnv service.

Tab. 10.6 *SimEnv include files and link scripts*

File / location	Used in / generated during	Explanation
simenv_mod_ [f | c | cpp].lnk $SE_HOME/lib	used in: stand alone	shell script to compile and link an interfaced model source code for experiment performance If necessary copy it to $SE_WS and modify it
simenv_opr_ [f | c | cpp].lnk $SE_HOME/lib	used in: stand alone	shell script to compile and link a user-defined operator source code for experiment post-processing If necessary copy it to $SE_WS and modify it
simenv_mod_ [f | c].inc $SE_HOME/inc	used in: interfaced Fortran/C/C++ models	ASCII include file for an interfaced model source code to define SimEnv interface functions
simenv_mod_auto_ [f | c].inc $SE_HOME/inc	used in: interfaced Fortran/C/C++ models	ASCII include file for an interfaced model source code to define SimEnv interface functions and to declare auxiliary variables for the semi-automated model interface
simenv_opr_ [f | c].inc $SE_HOME/inc	used in: user-defined Fortran/C/C++ operators	ASCII include file for a user-defined operator source code to define SimEnv interface functions
<model>_ [f | c | py | sh | as].inc $SE_WS	generated during: experiment preparation (only for service run, not for service re-start, only for auto_interface ≠ no in <model>.cfg)	ASCII include file for semi-automated model interface The files can be used directly in the interfaced model source code (for Fortran, C/C++, and Python) or as a dot script in <model>.run (for the shell script and ASCII interface)

10.4 User-Defined Files and Shell Scripts, Temporary Files

Tab. 10.7 lists the mandatory or optional shell scripts and files the user has to provide for running SimEnv services.

Tab. 10.8 lists the temporary or permanent files that are created during a SimEnv service.

Tab. 10.7 *User files and shell scripts to perform any SimEnv service*
***(*)**: make sure by the Unix / Linux command chmod u+x <file>*
that a file <file> has execute permission

Shell script / file (in the current workspace $SE_WS)	Explanation	Exist status	For more information see Section
<model>.cfg	ASCII user-defined general configuration file	optional	10.1
<model>.mdf	ASCII user-defined model (variables) description file	mandatory	5.1
<model>.edf	ASCII user-defined experiment description file	mandatory	6.1
<model>.mac	ASCII user-defined macro description file	optional	8.7
<model>.odf	ASCII user-defined operator description file	optional	8.5.4
<model>.gdf	ASCII user-defined GAMS model output description file	for GAMS models mandatory	5.7.2
<model>.run **(*)**	model shell script to wrap the model executable	mandatory	7.6
<model>.rst **(*)**	model shell script to prepare single model run restart	optional	7.6
<model>.ini **(*)**	model shell script to prepare simulation experiment additionally to standard SimEnv preparation	optional, for Python, Java, Matlab and GAMS models mandatory and standardized	7.6
<model>.end **(*)**	model shell script to wrap up simulation experiment	optional, for GAMS models mandatory and standardized	7.6
<model>.lnk **(*)**	model shell script to link an interfaced C/C++/Fortran model. Used in the course of experiment preparation for experiment run (not re-start) if a semi-automated model interface (auto_interface ≠ no) was declared in <model>.cfg for the appropriate programming languages. Can also be used stand alone for non-semi-automated model interface. Is normally based on $SE_HOME/lib/simenv_mod_[f | c | cpp].lnk	optional	5.10
<model>_ [dis | par | seq]_ [aix | linux].jcf	user-specific job control file to submit a job by the load leveler in distributed / parallel / sequential mode	optional	7.6
<model>_opt_ options.txt	user-specific control and option file for experiment type optimization	optional	6.6.1
<model>.err <simenv_ run_char>	touch / create this file in the model or in <model>.run as an indicator to stop the complete experiment after <model>.run has been finished for the single model run <simenv_run_int>	optional	7.6
usr_opr_<opr> **(*)** (in the opr_directory according to <model>.cfg)	executable for user-defined operator <opr>	optional	8.5

Tab. 10.8 *Files generated during performance of SimEnv services*
For the current workspace $SE_WS see Tab. 10.14.

File / location	Generated in	Explanation
Permanent files		
<model>.smp $SE_WS	experiment preparation (all but optimization) experiment performance (optimization)	ASCII sample input file for the run ensemble derived from <model>.edf Record no. n+1 corresponds to single run no. n. Column no. m of each record is the sampled value for experiment factor no. m in the edf-file
<model>_ [f | c | sh | as].inc and <model>_py.py $SE_WS	experiment preparation (if auto_interface ≠ no in <model>.cfg)	ASCII include files / dot scripts for semi-automated model interface
<model>.out <simenv_run_char> .[nc | ieee] model out_directory	experiment performance (if out_separation = yes in <model>.cfg)	model output of run number <simenv_run_int> of the experiment to be processed by the experiment post-processor
<model>.outall .[nc | ieee] model out_directory	experiment performance (if out_separation = no in <model>.cfg)	model output of all runs of the experiment to be processed by the experiment post-processor
<model>.elog $SE_WS	experiment performance	ASCII minutes file of **e**xperiment performance (simenv.run and all successive simenv.rst)
<model>.mlog $SE_WS	experiment performance	ASCII minutes file of **m**odel interface functions performance (simenv.run and all successive simenv.rst) <model>.mlog is organized single run by single run
<model>.nlog $SE_WS	experiment performance	ASCII minutes file of **n**ative - model specific experim. prepar. by <model>.ini - single runs model output by <model>.run - single run restart preparation by <model>.rst - model specific experim. wrap-up by <model>.end performances, redirected from terminal (simenv.run and all successive simenv.rst) <model>.nlog is organized single run by single run
run<simenv_run_char> $SE_WS	experiment performance (only for GAMS models)	sub-directory for GAMS model performance that are kept according to the sub-keyword 'keep_runs' in <model>.gdf
<model>.olog $SE_WS	experiment performance (only for experiment type optimization)	ASCII minutes file of **o**ptimization experiment performance
<model>.fct $SE_WS	experiment performance (only for experiment types optimization and Monte Carlo with stopping rule)	ASCII file of function values. Record no. n+1 corresponds to single run no. n.
<model>. killed<simenv_run_char> $SE_WS	experiment performance	indicator file that in <model>.run a process was killed by the shell script simenv_kill_process due to CPU time exceeding

File / location	Generated in	Explanation
<model>.res <simenv_res_char> .[nc | ieee | ascii] postproc out_directory	experiment post-processing	output file of an experiment post-processing session
<model>.inf <simenv_res_char> .[ieee | ascii] postproc out_directory	experiment post-processing	output structure description file of an experiment post-processing session
Temporary files (do not delete during performing the corresponding service)		
<model>. out<simenv_run_char> .[nc | ieee] model out_directory	experiment performance (if out_separation = 'no' in <model>.cfg)	if the experiment is performed by the load leveler in distributed or parallel mode
<model>. as<simenv_run_char> $SE_WS	experiment performance (only for simenv_get_as)	ASCII file with all factor names and their adjusted values
asa_opt asa_out asa_usr_out $SE_WS	experiment performance (only for experiment type optimization)	auxiliary files for experiment type optimization
run<simenv_run_char> sub-direct. of $SE_WS	experiment performance (only for Mathematica and GAMS models)	sub-directory for Mathematica and GAMS model performance
<model>_ [pre | main | post].inc $SE_WS	experiment performance (only for GAMS models)	auxiliary files <model> = GAMS main and all interfaced sub-models
<model>.res00.nc $SE_WS	experiment post-processing	NetCDF representation of the current result for visualization during experiment post-processing (only for value "yes" of sub-keyword 'visualization' in <model>.cfg)
simenv_get_experiment .exc $SE_WS	experiment post-processing	auxiliary file for operator get_experiment
simenv_*.tmp $SE_WS	all services	auxiliary files

Fig. 10.1 sketches usage of main SimEnv user shell scripts and files in the course of model interfacing, experiment preparation and performance, experiment post-processing, and visual evaluation of post-processed results.

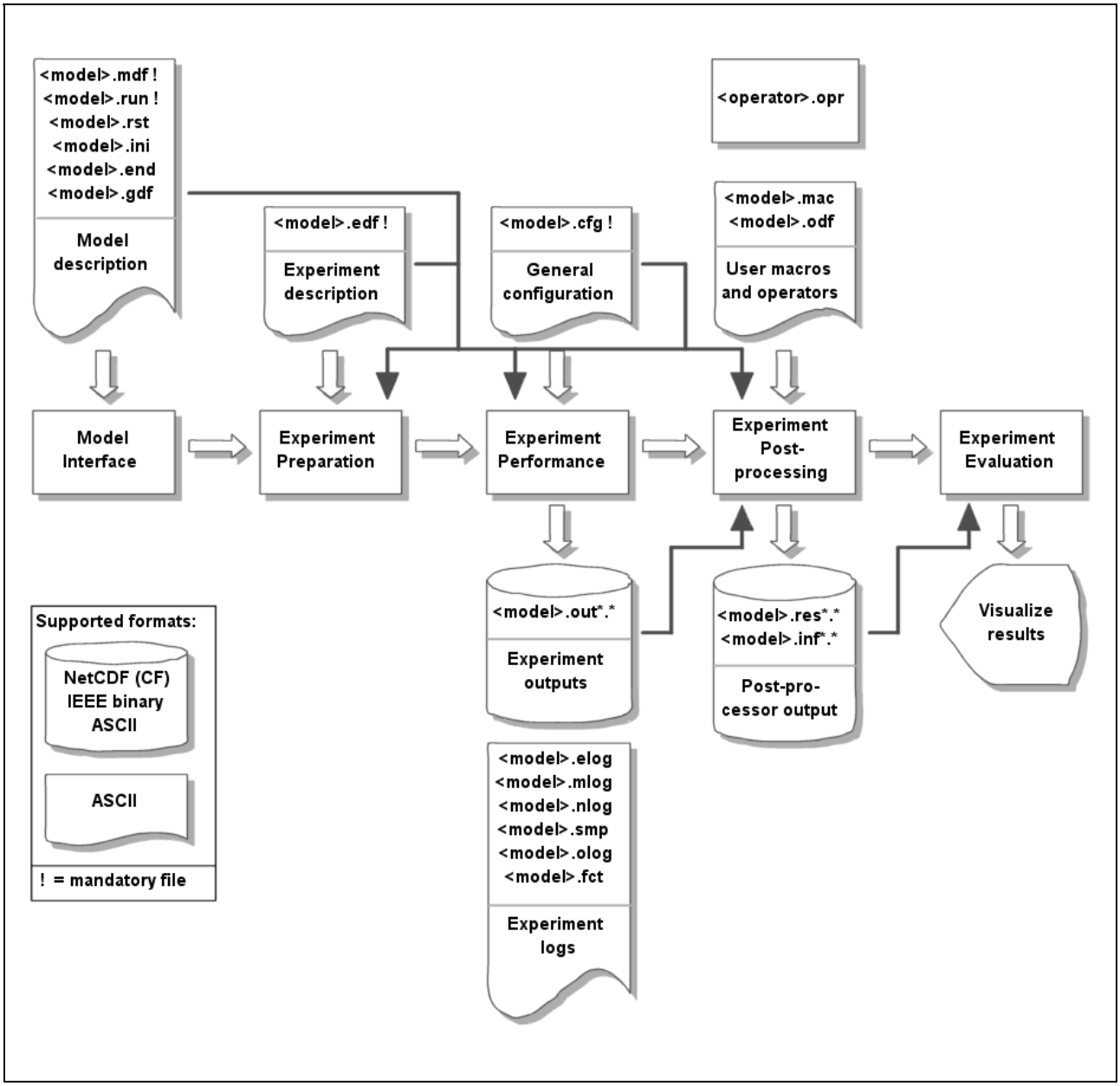


Fig. 10.1 *SimEnv user shell scripts and files*

10.5 Built-In Names

SimEnv has a number of built-in model output variable, shell script variable and coordinate names that cannot be used for corresponding user-defined names.

Tab. 10.9 lists the built-in (pre-defined) model variables that are output during experiment performance to SimEnv model output structures and are available in experiment post-processing without defining them in the model output description file <model>.mdf and without using the corresponding model interface coupling functions simenv_put_* in the model.

Tab. 10.9 *Built-in model output variables*

Built-in model output variable name	Dimen-sionality	Extents	Data type	Meaning
sim_time	0		float	elapsed user time in seconds when perform-ing /usr/bin/time –p <model>.run

Tab. 10.10 lists the built-in (pre-defined) shell script variables that are defined / used by the model coupling interface dot scripts $SE_HOME/bin/simenv_*_sh and simenv_run_[mathematica | gams] and that are finally available in <model>.run.

Tab. 10.10 *Built-in shell script variables in <model>.run*

Built-in shell script variable name	Meaning
simenv_run_int	current run number as integer
simenv_run_char	current run number as 6-character string with leading zeros
factor_name	factor name for simenv_get_sh
factor_def_val	default factor value for simenv_get_sh
simenv_hlp_*	auxiliary variables

Tab. 10.11 lists the built-in (pre-defined) coordinates that are used in experiment post-processing when additional dimensions are generated by an operator.

Tab. 10.11 *Built-in coordinates for experiment post-processing*

Built-in coordinate name	Generated by operator	Meaning	Definition (cf. Tab. 11.6)
bin_mid	hgr hgr_e hgr_l	bin mid values	equidist_end <xx>(<yy>) 999999 with <xx> = first bin mid <yy> = bin width
bin_no	hgr hgr_e hgr_l	bin numbers	equidist_end 1(1)999999
incr	lin_abs lin_rel sens_abs sens_rel sym_abs sym_rel	increment values	dependent on experiment description and operator arguments
index	maxprop maxprop_l minprop minprop_l	index number	equidist_end 1(1)999999
run	ens	run numbers	equidist_end 1(1)999999

Built-in coordinate name	Generated by operator	Meaning	Definition (cf. Tab. 11.6)
sign	sens_abs sens_rel	signs of incremental change: - 1: -ε +1: +ε	equidist_end -1(2)1
stat_measure	stat_full stat_red	basic statistical measures: 1: deterministic case 2: minimum 3: maximum 4: mean 5: variance positive distance from mean of confidence measure … 6: …. 1 7: …. 2 8: median 9: quantile of quantile value 1 10: quantile of quantile value 2	equidist_end 1(1)10
factor_sequ	morris lin_abs lin_rel sens_abs sens_rel sym_abs sym_rel	sequence of factors: 1: 1st factor in edf-file 2: 2nd factor in edf-file ...	equidist_end 1(1)999999
<factor_name>	behav	factor values	dependent on experiment description and operator arguments

10.6 Case Sensitivity

As stated in Tab. 10.12 all names used in SimEnv are case insensitive. Internally, they are mapped on a lowercase representation and this lowercase representation is used also for model and/or experiment post-processor output files in NetCDF, IEEE and/or ASCII format.

Tab. 10.12 *Case sensitivity of SimEnv entities*

Where?	Entity	Case sensitivity	Example
overall	• model name	sensitive	`simenv.chk World_f`
user-defined files (cf. Section 11.1)	• keyword • name • sub-keyword	insensitive	`experiment END_RUN last`
	• information <value>	insensitive	`experiment end_run LAST` `general descr This is ...`

model interface	• variable and factor name	insensitive	`iok=simenv_put_f('ATMO',`
			`atmo,atmo)`
			`factor_name='P1'`
			`factor_value=1.`
			`. $SE_HOME/bin/simenv_get_sh`

Where?	Entity	Case sensitivity	Example
experiment post-processing	• optional result description and unit	sensitive	`Energy [kW] = my_opr(atmo)`
	• character arguments of user-defined operators	sensitive	`char_test('arg11' , 'Arg21' , atmo)`
	• variable and factor name • operator name • number • macro name • macro identifier _m	insensitive	`3e-6*exp(atmo) +` `3E-6*EXP(ATMO)`
	• character arguments of built-in operators with pre-defined values (cf. Tab. 15.10)	insensitive	`count('ALL' , atmo)`
	• character arguments of built-in operators without pre-defined values	check Tab. 15.10	`get_table_fct('MyFile.dat' , atmo)` `get_experiment('../' , 'Model_f' ,' ', atmo)`
Exceptions			

information <value> in user-defined files	• <directory> and <file_name>
	- for <sub-keyword> = '<string>_directory'
	- and in <val_list>
	• <value> for <sub-keyword> = ['descr' | 'unit'])
		sensitive	`model out_directory MyDir`
			`factor p1 sample file MyF`
			`factor p1 unit kWh`

<model>.edf (for Mathematica models)	• <factor_name>	sensitive as in the Mathematica model	`factor p1 sample list 1,2,3` `factor P1 sample list 3,4,5`
<model>.gdf (for GAMS models)	• GAMS model file name	sensitive	`model sub_m1 type sub` `model sub_M1 type sub`

10.7 Numerical Nodata Representation

For model output with the SimEnv model coupling interface functions and for experiment post-processor output the following data type specific nodata values are used to represent undefined (unwritten) model output and undefined post-processor output:

Tab. 10.13 *Data type related nodata values*

SimEnv data type (cf. also Tab. 5.4)		Nodata value
byte	int*1	= 127
short	int*2	= 32767
int	int*4	= 2147483648
float	real*4	≥ 3.4E+38
double	real*8	≥ 1.79D+308

10.8 Operating System Environment Variables

The following operating system environment variables are used by SimEnv.
Additionally, make sure that in the shell the noclobber option is **not** set.

Tab. 10.14 *Environment variables*

Environment variable	Meaning	Definition status	Explanation
SimEnv access settings Set by the user Used within all SimEnv services			
SE_HOME	SimEnv home directory	mandatory	Value = has to be defined by the user For values check Tab. 3.3 and Tab. 15.1. Optionally, include $SE_HOME/bin into the PATH environment variable to access to a SimEnv service without prefixing it by $SE_HOME/bin/
DISPLAY	machine / screen that the X11-system uses for displaying windows	optional	Value = machine dependent has to be defined at PIK by the user only for visualization matters in SimEnv services simenv.res and simenv.vis.
Internal settings Set automatically by SimEnv Set within all SimEnv sevices			
SE_GUI	identifier for GUI / non-GUI version	--	for all SimEnv services Value = [yes | <nil>]
SE_MOD	model name	--	for all SimEnv services Value = <model>
SE_OS	operating system specification	--	for all SimEnv services Value = [AIX | LINUX]
SE_WS	current SimEnv workspace	--	for all SimEnv services Value = <directory>
SE_RUN	run number of a single run	--	for <model>.[run | rst] Value = <simenv_run_int>
SE_RUN1	first single run of an experiment	--	for <model>.[run | rst] Value = [yes | no]

To perform SimEnv, make sure that paths to the directories of the programs as specified in Tab. 10.15 below are included in the environment variable PATH:

Tab. 10.15 *Programs to include in the environment variable PATH*

Program	Usage	Include in PATH
python	python interpreter	mandatory
ncdump	dump NetCDF files	mandatory
gams	GAMS modelling system	optional, only for running GAMS models

Program	Usage	Include in PATH
java	Java	optional, only for running Java models
matlab	Matlab interpreter	optional, only for running Matlab models
MathKernel	Mathematica interpreter	optional, only for running Mathematica models

Additionally, PATH is prefixed by $SE_WS (see Tab. 10.14 above) internally by all SimEnv services. Keep in mind to specify a PYTHONPATH environment variable dependent on interfaced Python models. PYTHONPATH is prefixed by $SE_WS and $SE_HOME/bin is appended to PYTHONPATH internally by all SimEnv services.
For linking and running Fortran and C/C++ models and operators the environment variables PATH and LIBRARY_PATH have to be defined accordingly.

11 Structure of User-Defined Files, Coordinate Transformation Files, Value Lists

Basic information to describe general control settings of SimEnv, model output variables, the experiment itself, macros and user-defined operators as well as GAMS model specific information is stored in user-defined files. They are ASCII files and have a common structure that is described in this chapter. Additionally, coordinate transformation files are described and value lists are defined in general.

11.1 General Structure of User-Defined Files

All user-defined files listed in Tab. 11.1 have the same structure. They are ASCII-files with the following record structure:

{ <sep> } <keyword> <sep> { <name> <sep> } <sub-keyword> <sep> <value> { <sep> }

with

- <name> is the name of a
  - model output variable
  - GAMS model source file
  - experiment factor
  - coordinate
  - user-defined operator or
  - macro

  Declaration of <name> depends on the related keyword <keyword>
- <keyword> is a string
  Normally, more than one lines with differing sub-keywords belong to one “keyword-block”.
- <sub-keyword> is a string
  Sub-keywords are defined only in relation to the user file and the keyword under consideration.
- <value> = <substring> { <sep> <substring> ... }
  is a string with user file, keyword and sub-keyword related information.
- <sep> is a sequence of white spaces

Sequence of keyword and sub-keyword lines can be arbitrary. For reasons of readability it is recommended to use a block structure like in the Example 11.2 below. Sequence of names in the separated name spaces (name spaces of coordinates, model output variables, experiment factors, user-defined operators, macros) during processing is determined by the sequence the name occur the first time in the appropriate user file.
Lines consisting only from separator characters as well as lines starting with a # as the first non-separator character are handled as comment lines. For case sensitivity of the contents of user-defined files check Tab. 10.12 on page 137.

Tab. 11.1 *User-defined files with general structure*

File	Contents	See description in Section	See description on page
<model>.cfg	general configuration file	10.1	125
<model>.mdf	model output description file	5.1	25
<model>.edf	experiment description file	6.1	53
<model>.odf	operator description file	8.5.4	117
<model>.mac	macro description file	8.7	119
<model>.gdf	GAMS description file	5.7.2	39
arbitrary file name	coordinate transformation file	11.2	143

The following restrictions hold for user-defined files:

Tab. 11.2 *Constraints in user-defined files*
***(*)**: with the exception for GAMS model source code file names*

Element	Constraints
line length	max. 160 characters
<name>	max. 20 characters
	(*) first character has to be a letter
	(*) must not end on _m
	(*) must not contain elemental operators and characters . and : (cf. Tab. 8.3 on page 86)
<value>	for sub-keyword = 'descr' without <name>: max. 512 characters (total sum over all lines)
	for sub-keyword = 'descr' with <name>: max. 128 characters
	for sub-keyword = '<string>_directory': max. 100 characters (for the resulting resolved directory string, directory can contain operating system environment variables)
	for sub-keyword = 'unit': max. 32 characters

Tab. 11.3 lists the reserved (forbidden) names and file names that cannot be declared in user-defined files.

Tab. 11.3 *Reserved names and file names in user-defined files*

Element	Reserved (forbidden) names
<name> (with the exceptione of GAMS model source code file names)	built-in model output variables according to Tab. 10.9
	built-in coordinates according to Tab. 10.11
	special keywords in <model>.edf for behavioural analysis: [default | file]
<file_name>	see Section 11.3

The **line type** in the description table for a user-defined file specifies whether a keyword / sub-keyword combination can be omitted.

Tab. 11.4 *Line types in user-defined files*

Abbre-viation	User-defined file	Explanation
m	all files	**m**andatory
o	all files	**o**ptional
c1	<model>.mdf keyword ‘variable’ sub-keyword [‘coords’ | ‘index_extents’]	**c**onditional **1**: forbidden for variables with dimensionality = 0 mandatory for variables with dimensionality > 0
c2	<model>.mdf keyword ‘variable’ sub-keyword ‘coord_extents’	**c**onditional **2**: forbidden for variables with dimensionality = 0 optional for variables with dimensionality > 0
c3	<model>.edf keyword ‘factor’ sub-keyword ‘sample’	**c**onditional **3**: mandatory for experiment type = Monte Carlo analysis forbidden for experiment type = local sensitivity analysis conditional for experiment type = behavioural analysis
c4	<model>.edf for Monte Carlo analysis keyword ‘factor’ sub-keyword ‘sampling’	**c**onditional **4**: mandatory for sample = distr ... forbidden for sample = file ...
a	<model>.edf for behavioural analysis keyword ‘factor’ sub-keyword ‘sample’	**a**lternatively: either mandatory for all experiment factors or forbidden for all experiment factors
f	<model>.edf for local sensitivity analysis keyword ‘factor’ sub-keyword ‘sample’	**f**orbidden

```
mac                 descr     This is a macro description file
mac                 descr     for the SimEnv User Guide

macro    pol_atmo   descr     atmo outside polar reg., final time, level 1
macro    pol_atmo   unit      without
macro    pol_atmo   define    atmo(c=84:-56,*,c=1,c=20)

macro    m1         define    avg(atmo_g(c=11:20))
...
```

Example 11.1 *Structure of a user-defined file*

11.2 Coordinate Transformation File

Some post-processing operators (currently, get_data and get_experiment) enable access to external data. They derive from an operator argument a multi-dimensional result that has to be equipped - as usual in SimEnv experiment post-processing - with a coordinate assignment. By applying these operators it can be necessary to define or transform a coordinate description for the operator result that fits the result to the current model and/or experiment under consideration. The same is true for the operator regrid which is used to assign new coordinates to a result. The following cases can be distinguished:

- A dimension of the result does not have a coordinate assignment. A coordinate has to be assigned to this dimension.
- A coordinate description of the result has to be modified in a way that it matches with a defined coordinate of the model / experiment under consideration.
- A coordinate description of the result has to be incorporated with and/or without modifications into the coordinate set of the model / experiment under consideration.

Coordinate transformations for results in the course of the operator's performance are supported in SimEnv by a coordinate transformation file that is assigned to the operator result as an argument of the operator. Coordinate transformation files follow the same syntax rules as all other user-defined files (cf. Section 10.1).

Tab. 11.5 *Elements of a coordinate transformation file*

keyword	name	sub-keyword	Line type	Max. line nmb.	value	Explanation
general	<nil>	descr	o	any	<string>	general transformation description
modify	<original_coordinate_name>	rename	o	1	<new_name>	renames original coordinate
		position_shift	o	1	<real_val>	shifts all values of the original coordinate by the specified value <position_shift_val>
		values_shift	o	1	<int_val>	shifts the result values on the original coordinate by the specified positions <values_shift_val>
		values_add	o	1	<val_list>	defines <values_shift_val> values to add to the coordinate values (for syntax see Tab. 11.6)
assign	[<original_coordinate_name> | <coordinate_nmb>]	coord	o	1	<co_name>	assign to the dimension with coordinate number <coordinate_nmb> (only for operator get_data('ascii',…) and/or <original_coordinate_name> (else) an already defined coordinate or a coordinate defined by the keyword 'coordinate'
		coord_extent	o	1	<co_val_1>: <co_val_2>	assigns start and end coordinate value to the dimension of the result under consideration
coordinate	<new_coordinate_name>	descr	o	1	<string>	coordinate axis description
		unit	o	1	<string>	coordinate axis unit
		values	o	1	<val_list>	strictly monotonic sequence of coordinate values (for syntax see Tab. 11.6)

To Tab. 11.5 the following additional rules and explanations apply:

- For the description of **line type** check Tab. 11.4 on page 143.
- With the sub-keyword '**values_shift**' result values can be shifted on the corresponding coordinate by <values_shift_val> coordinate values. Consequently, <values_shift_val> coordinate values have to be appended at the end of the coordinate for a positive value of <values_shift_val> and/or have to be inserted at the begin of the coordinate for a negative value of <values_shift_val>. Coordinate values that are obsolete because of this shift are removed from the coordinate definition.
  For a coordinate that is defined with equidistant coordinate values the extent of the coordinate is specified automatically by simply applying the equidistant rule for this coordinate.
  For a coordinate with non-equidistant coordinate values the coordinate values necessary for the coordinate extension are defined by the sub-keyword '**values_add**'.
  If both '**position_shift**' and '**values_shift**' are specified for one coordinate, firstly position shift is applied to the coordinate and then the additional coordinate values from values_shift are added the the coordinate without applying the position_shift value.
- Coordinate numbers **<coordinate_nmb>** are integers counting from 1.

- For the sub-keyword '**coord_extent**' the same rules apply as for the sub-keyword 'coord_extents' from the model output description file <model>.mdf.
- For the keyword '**coordinate**' the same rules apply as for the keyword 'coordinate' from the model output description file <model>.mdf.
- Coordinates are incorporated additionally into the original coordinate set only for the current result.

Unlike all other user-defined files coordinate transformation files cannot be checked by the SimEnv service simenv.chk or when starting the service simenv.res.

Having a model output variable definition as in Example 5.1 on page 31 and
assuming address_default = coordinate in <model>.cfg
Assume the experiment layout in Fig. 4.4 (c) on page 18 and
the corresponding experiment description file from Example 6.3 (3c) on page 58.

Additionally, assume another experiment with a model named model and there
model output variables modvar1 and modvar2 that are defined for the following coordinates:

dimension	coordinate name	coordinate definition
1	dim1	list 1,10,100,1000
2	dim2	equidist_end 2(2)20
3	dim3	equidist_end 3(3)30
4	dim4	equidist_end 4(1)43
5	dim5	equidist_end 5(1)50

Further, assume the coordinate transformation file model.ctf as

```
general                    descr             example of a coordinate
general                    descr             transformation file

modify          dim1       rename            new1
modify          dim1       position_shift    3.
modify          dim1       values_shift      +2
modify          dim1       values_add        list 1006,1009

modify          dim3       values_shift      -3

assign          dim4       coord             lat
assign          dim4       coord_extent      88.:-68.

assign          dim5       coord             new2
assign          dim5       coord_extent      50.:5.

coordinate      new2       descr             new coordinate
coordinate      new2       values            equidist_end 50(-1)5
```

In experiment post-processing the result of the expression

```
get_experiment('mydir','model','model.ctf',modvar1+modvar2)
```

is a 5-dimensional data structure with

dimension	coordinate name	coordinate definition	coordinate use
1	new1	list 103,1003,1006,1009	= coordinate definition
2	dim2	equidist_end 2(2)20	= coordinate definition
3	dim3	equidist_end -6(3)21	= coordinate definition
4	lat	equidist_end 88(-4)-88	equidist_end 88(-4)-68
5	new2	equidist_end 5(1)50	= coordinate definition

Example 11.2 *Coordinate transformations by a transformation file*

11.3 ASCII Data Files and Value Lists

ASCII data files {<directory>/}<file_name> are used in SimEnv as an element for the specification of value lists (see below), optionally in experiment description files to get sampling information, and in post-processing operators.

The following rules and restrictions are valid for {<directory>/}<file_name>:

- The <directory> path can contain operating system environment variables ($...)
- If <directory> is specified in a relative manner (./…) it relates to the current workspace
- <file_name> must not be one of the SimEnv file names according to Tab. 10.7 and Tab. 10.8
- For the file:
  - Has to be an ASCII file
  - Can be a multi-record file
  - Max. record length is 1000 characters
  - Values in a record are separated from each other by white spaces or comma
  - A series of connected (running) separators is treated as a single separator
  - Record end is handled as a separator
  - Records formed only from white spaces or records starting with the first non-white space character # are handled as comments

For variables, coordinates and experiment factors value lists are supplied by the value-item in user-defined files. Value lists describe a sequence of values together with an order. The number of described values has to be greater than 1. Value lists may be restricted to strictly monotonic sequences. They follow the syntax rules in Tab. 11.6.

Tab. 11.6 *Syntax rules for value lists*

Value-list type	Syntax		Explanation
explicit	list	<real_val$_1$> ,..., <real_val$_n$>	explicit list of values same syntax rules as for one record of a file with a value list (see above)
by reference	file	{<directory>/}<file_name>	file {<directory>/}<file_name> contains the explicit value list
implicit with begin element increment end element	equidist_end	<real_val$_1$> (<real_val$_2$>) <real_val$_3$>	description of an equidistant list of values with begin value <real_val$_1$> increment <real_val$_2$> end value <real_val$_3$> <real_val$_1$> ≠ <real_val$_3$> <real_val$_2$> ≠ 0. Number of resulting values have to be > 1
implicit with begin element increment number of values	equidist_nmb	<real_val$_1$> (<real_val$_2$>) <int_val>	description of an equidistant list of values with begin value <real_val$_1$> increment <real_val$_2$> number of values <int_val> <real_val$_2$> ≠ 0. <int_val> > 1

Value-list type	Syntax	Explanation
implicit with begin element number of values end element	equidist_ivl <real_val$_1$> (<int_val>) <real_val$_2$>	description of an equidistant list of values with begin value <real_val$_1$> number of values <int_val> end value <real_val$_2$> <int_val> > 1 <int_val> - 2 values are placed within the interval [begin_value : end_value]

1	`list 3, 5, 7, 9, 11`	describes the five values 3, 5, 7, 9, and 11
2	`equidist_end 3 (2) 11`	is equivalent to 1
3	`equidist_end 3 (2) 11.9`	is equivalent to 1
4	`equidist_nmb 3 (2) 5`	is equivalent to 1
5	`equidist_ivl 3 (5) 11`	is equivalent to 1
6	`file my_values.dat`	is equivalent to 1 with `my_values.dat =3, , 5, 7 9, 11`
7	`equidist_end 11 (-2) 3`	differs from 1 – 6: values are identical, ordering sequence differs

Example 11.3 *Examples of value lists*

12 Model and Experiment Post-Processor Output Data Structures

This chapter summarizes information on available data structures for model and experiment post-processor output. SimEnv supports several output formats from the experiment and the post-processor. NetCDF is a self-describing data format and can be used for model and post-processor output. Another format specifications for both outputs is IEEE compliant binary format and ASCII for post-processor output. This chapter describes all the used data structures.

Dependent on the specification of the supported experiment post-processor output formats in <model>.cfg model output can be stored in NetCDF format and post-processor output in NetCDF, IEEE or ASCII format.
During experiment performance model output is written either to single output files <model>.out<simenv_run_char>.[nc | ascii] per experiment single run or to a common output file <model>.outall.[nc | ieee] for all single runs from the experiment run ensemble. Output to single files or a common file depends on specification of the value for the sub-keyword 'out_separation' in <model>.cfg. <simenv_run_char> is a six-digit placeholder for the corresponding single run number.
During experiment post-processing output and structure of results is written to <model>.res<simenv_res_char>.[nc | ieee | ascii] and <model>.res<simenv_res_char>.[ieee | ascii]. <simenv_res_char> is a two-digit placeholder for the number of the result file. It ranges from 01 to 99.
For IEEE and ASCII model output and experiment post-processor output formats, multi-dimensional data is organized in the Fortran storage model (cf. Section 15.7 - Glossary).
Use the SimEnv service command simenv.dmp for browsing model and result output files. See Tab. 10.4 for more information.

12.1 NetCDF Model and Experiment Post-Processor Output

The intention for supplying NetCDF format for model and experiment post-processor output is to provide the possibility to generate self-describing, platform-independent data files with metadata that can be interpreted by subsequent visualization techniques. The conventions applied for SimEnv represent a compromise between existing standards and the metadata requirements for a flexible and expressive visualization that is adapted to the requirements of the specific data sets of concern. SimEnv follows the NetCDF Climate and Forecast (NetCDF CF) metadata convention 1.0. Currently, SimEnv supports only up to 4-dimensional NetCDF output during experiment and post-processor performance.

In principle, any NetCDF file can be viewed by the NetCDF service program

ncdump <NetCDF_file>

Model output data types as declared in the model output description file <model>.mdf are transferred into NetCDF data types automatically (cf. the Table below). By default, post-processor output data is of type float.

Tab. 12.1 *NetCDF data types*

SimEnv data type (cf. Tab. 5.4)	NetCDF data type
byte	NF_BYTE
short	NF_SHORT
int	NF_INT
float	NF_FLOAT
double	NF_DOUBLE

12.1.1 Global Attributes

The global attributes used in SimEnv from the CF standard are :institution and :Conventions. In addition, the following global attributes are defined for model and post-processor output:

Tab. 12.2 *Additional global NetCDF attributes*

Name	Value	Data type
:creation_time	<YYYY-MM-DD HH:MM:SS>	char
:model_name	<model>	char
:model_description	model output description according to <model>.mdf	char
:model_description_file	{<directory>/}<model>.mdf	char
:experiment_type	experiment type according to Tab. 6.1	char
:experiment_description	experiment description according to <model>.edf	char
:experiment_description_file	{<directory>/}<model>.edf	char
:number_of_runs	<number of runs>	int

12.1.2 Variable Labelling and Variable Attributes

For NetCDF variables, two cases of labelling are distinguished:

- If
  - during experiment performance for a SimEnv model output variable or
  - during post-processing for a SimEnv result

  one of its coordinates spans the entire range of definition, the already defined coordinate definition is used.
- Otherwise, an additional coordinate
  <variable_name>-<coordinate_name>
  is defined, where the NetCDF variable depends on. The additional variable is described in the dimension and data part of the NetCDF file. Additionally, the SimEnv specific attribute
  index_range_<original_coordinate_name> (see Tab. 12.4)
  is assigned to such a NetCDF variable.

The following variable attributes are used according to the CF-1.0 standard:

Tab. 12.3 *Variable NetCDF attributes*

Name	Value	Data type
<variable_name>:standard_name	[<coordinate_name> | <predef_coordinate_name> | <predef_var_name> | <factor_name> | <variable_name> | <result_name>]	char
<variable_name>:long_name	[<coordinate_description> | <predef_coordinate_description> | <predef_variable_description> | <factor_description> | <variable_description> | <result_applied_operator_sequence>]	char

Name	Value	Data type
<variable_name>:unit	[<coordinate_unit> | <predef_coordinate_unit> | <predef_variable_unit> | <factor_unit> | <variable_unit> | <result_unit>]	char
<variable_name>:missing_value	<variable type-depending missing value>	type-dep.
<variable_name>:axis (single coordinate variables only)	[X | Y | Z | T | bin_no | run | …]	char
<variable_name>:coordinates (multi-dimensional coordinate variables only)	<par1_lon> <par1_lat>	char
<variable_name>:_Fillvalue	<variable type-depending fill value>	type-dep.

- For experiment post-processor output, the **:standard_name** attribute simply counts the number of applied operations because the result name of an arbitrary operation is not known in general. For that reason, the :long_name attribute would re-sample the :standard_name attribute and it is used instead to provide the complete description of the applied operator sequence without defining an additional attribute.
  If macros are included, these are resolved and elementary operations are included only.
- For the **:axis** attribute of a coordinate variable exist defaults.
  For each post-processor result, the first coordinate is assumed to be the „X-axis“, the second and third coordinate are assumed to represent the „Y-“ and „Z-axis“, and the fourth dimension is time T.
  For model results, these attribute values are assigned to coordinate variables describing geographical longitude, geographical latitude, level or height and time. In case other coordinate names are used, these are simply also used for the axis attribute.
- The **:unit** attribute is actually estimated for model output only depending on the description of the corresponding sub-keywords for the keyword ‘variable’ in the <model>.mdf file. For post-processing output, it is only used as a placeholder and not calculated from the applied operator sequence so far.
- The **:coordinates** attribute serves to define coordinates depending on other ones and so to allow coordinate transformations. Actually, this attribute is not used.
- Actually, the **:_Fillvalue** attribute is not applied to coordinate variables. It is identically to the :missing_value attribute but open for other definitions.

For visualization requirements, the following additional variable attributes have been defined for SimEnv:

Tab. 12.4 *Variable NetCDF attributes for visualization*

Name	Value	Data type
<variable_name>:monotony (coordinate variables only)	[increasing | decreasing | none]	char
<variable_name>:coo_type	[1 | 2]	integer
<variable_name>:data_range	<min> <max>	char
<variable_name>:index_range_<coordinate>	<min_index> <max_index>	int
<variable_name>:simenv_data_kind	[predefined model output variable | model factor | model output variable | postproc_result]	char
<variable_name>:var_representation	[positions | connections] or both	char
<variable_name>:grid_shift	<shift_x> <shift_y>	real, dimension(2)
<variable_name>:north_pole	<lon_pole> <lat_pole>	real, dimension(2)

- The **:monotony** attribute is applied to coordinate variables only and estimated from the coordinate values as defined in the <model>.mdf file. During post-processing additional coordinates can be generated for which no monotony may be estimated. In such cases, the attribute is set to “none”.
- The **:coo_type** attribute describes the grid representation of a given coordinate. A value of 1 indicates that all coordinate values are provided explicitly (suitable, e.g., for irregular grids). A value of 2 indicates a regular grid and a coordinate representation by its start value, increment and end value.
- The **:data_range** attribute provides the real range that is covered by the related variable in the recent NetCDF file.
- The **:index_range** attribute is used only in case a NetCDF variable does not cover the complete range of a coordinate and an additional coordinate was defined and assigned to this NetCDF variable. The index_range attribute describes that sub-space for which the NetCDF variable is defined. Range indices count from 1.
- The **:var_representation** attribute is introduced to specify what operations are allowed on the data.
- The **:grid_shift** attribute is actually still a placeholder for variables that are not defined in the centre of a grid box when quasi-regular grids are used.
- The **:north_pole** attribute can be used if rotated grids are applied.

12.2 IEEE Compliant Binary Model Output

IEEE compliant binary model output is written in records of fixed length to <model>.out<simenv_run_char>.ieee and/or <model>.outall.ieee. For the determination of the record length see below.

Sequence of data for each single run is as follows:
- Experiment factors as specified in <model>.edf
  Sequence as in <model>.edf
- Built-in (pre-defined) model output variables
  Sequence as in Tab. 10.9
- Model output variables
  Sequence as in <model>.mdf

Storage demand for each model output variable / factor is according to its dimensionality, extents and data type. Storage demand in bytes for each model output variable / factor is re-adjusted to the smallest number of bytes divisible by 8, where the data can be stored. Multi-dimensional data fields are organized in the Fortran storage model (cf. Section 15.7 - Glossary).
Data is stored in records with a fixed record length of minimum (512000 Bytes , re-adjusted storage demand in Bytes).
In <model>.outall.ieee each single run starts with a new record. Sequence of the single runs corresponds to the sequence of the single run numbers <simenv_run_int>. Consequently, data from default single run 0 is stored in the first and potentially the following records.

Having a model output description file as in Example 5.1 and an experiment description file as in Example 6.3 (3a) each single run is stored in the following way:

Factor / model variable	Extents	Data type	Storage demand [Byte]	Storage demand re-adjusted [Byte]
p1	1	float	4	8
p2	1	float	4	8
sim_time	1	float	4	8
atmo	45 x 90 x 4 x 20	float	1.296.000	1.296.000
bios	36 x 90 x 20	float	259.200	259.200
atmo_g	20	int	80	80
bios_g	1	int	4	8

				1.555.312

One single run needs 1.555.312 : 512.000 = 3+1 records with a fixed length of 512.000 Bytes. Remaining bytes in the last record are undefined.

Example 12.1 *IEEE compliant model output data structure*

12.3 IEEE Compliant Binary and ASCII Experiment Post-Processor Output

For IEEE and ASCII experiment post-processor output result information is stored in two files:

- <model>.res<simenv_res_char>.[ieee | ascii] holds the result dynamics
- <model>.inf<simenv_res_char>.[ieee | ascii] holds structure and coordinate information

The IEEE post-processor output files <model>.res<simenv_res_char>.ieee and <model>.inf<simenv_res_char>.ieee are unformatted binary files with IEEE float / int number representation, while for the ASCII post-processor version <model>.res<simenv_res_char>.ascii and <model>.inf<simenv_res_char>.ascii formatted ASCII files are used. Files for both output file formats have for each result subsequently the following structure:

Record structure of <model>.inf<simenv_res_char>.[ieee | ascii] for each result:

result number 01:

record no. 1	max. 512 chars	result expression string
record no. 2	max. 128 chars	result description string
record no. 3	max. 32 chars	result unit string (or 1 space if unit is undefined)
record no. 4	10 int	dim ext(1) ... ext(dim) 0 ... 0
record no. 4	max. 20 chars	coordinate name of dimension 1
record no. 5	10 float	coordinate values of dimension 1 in records of 10 values (last record may have less values)
...		
record no. xxx	max. 20 chars	coordinate name of dimension dim
record no. xxx+1	10 float	coordinate values of dimension dim in records of 10 values (last record may have less values)

result number 02:

...

Record structure of <model>.res<simenv_res_char>.[ieee | ascii] for each result:

result number 01:

record no. 1 ...	10 float	in records of 10 values (last record may have less values): result_value(1) ... result_value(length_result)
		with length_result = $\prod_{i=1}^{dim} ext(i)$ for dim > 0
		= 1 else

result number 02:

...

The vector result_value is stored in the Fortran storage model (cf. Section 15.7 - Glossary). The nodata element for undefined result values is set to 3.4E38.

The Fortran code in Example 15.15 reads experiment post-processing ASCII output files <model>.res<simenv_res_char>.ascii and <model>.inf<simenv_res_char>.ascii in their general structure. In the examples-directory $SE_HOME/exa of SimEnv it is accompanied by the corresponding version for IEEE result output.

13 SimEnv Prospects

SimEnv development and improvement is user-driven. Here one can find a list of the main development pathways in future.

General
- Graphical user interface
- Portability to Windows-based systems
- Unique number representations for binary IEEE output of distributed models (big endians vs. small endians)

Model interface

Experiment preparation
- Experiment type uncertainty analysis with variance decomposition
- Experiment type stochastic analysis
- Monte Carlo analysis: sampling of correlated factors

Experiment performance
- Experiment performance for distributed models across networks
- Multi-file model output storage

Experiment post-processing
- Additional advanced operators (coarse, sort, categorical operators)
- Advanced uncertainty operators
- Flexible assignment of data types to operator results (currently: only float)
- Shared memory access for user-defined operators to avoid data exchange by external files

Visual experiment evaluation
- Advanced techniques for graphical representation of experiment post-processor output, especially for multi-run operators

14 References and Further Readings

Campolongo, F., Cariboni, J., Saltelli, A., Schoutens, W. (2005): Enhancing the Morris Method. In: Hanson, K.M., Hemez, F.M. (eds.): Sensitivity Analysis of Model Output. Proceedings of the 4th International Conference on Sensitivity Analysis of Model Output (SAMO 2004). Los Alamos National Laboratory, Los Alamos, U.S.A., 369-379
http://library.lanl.gov/cgi-bin/getdoc?event=SAMO2004&document=samo04-52.pdf

European Commission, Joint Research Centre – IPSC (2006): SimLab 3 Website
http://simlab.jrc.ec.europa.eu/

Flechsig, M. (1998): SPRINT-S: A Parallelization Tool for Experiments with Simulation Models. PIK-Report No. 47, Potsdam Institute for Climate Impact Research, Potsdam
http://www.pik-potsdam.de/research/publications/pikreports/.files/pr47.pdf

Flechsig, M., Böhm, U., Nocke, T., Rachimow, C. (2005): Techniques for Quality Assurance of Models in a Multi-Run Simulation Environment. In: Hanson, K.M., Hemez, F.M. (eds.): Sensitivity Analysis of Model Output. Proceedings of the 4th International Conference on Sensitivity Analysis of Model Output (SAMO 2004). Los Alamos National Laboratory, Los Alamos, U.S.A., 297-306
http://library.lanl.gov/cgi-bin/getdoc?event=SAMO2004&document=samo04-22.pdf

Gray, P., Hart, W., Painton, L., Phillips, C., Trahan, M., Wagner, J. (1997): A Survey of Global Optimization Methods. Sandia National Laboratories, Albuquerque, U.S.A.
http://www.cs.sandia.gov/opt/survey

Helton, J.C., Davis, F.J. (2000): Sampling-Based Methods.
In: Saltelli *et.al* (2000)

Iman, R.L., Helton, J.C. (1998): An Investigation of Uncertainty and Sensitivity Analysis Techniques for Computer Models. Risk Anal. 8(1), 71-90

Ingber, L. (1989): Very fast simulated re-annealing. Math. Comput. Modelling, 12(8), 967-973
http://www.ingber.com/asa89_vfsr.pdf

Ingber, L. (1996): Adaptive simulated annealing (ASA): Lessons learned. Control and Cybernetics, 25(1), 33-54
http://www.ingber.com/asa96_lessons.pdf

Ingber, L. (2004): ASA-Readme.
http://www.ingber.com/ASA-README.pdf

McKay, M.D., Conover, W.J., Beckman, R.J. (1979): A Comparison of Three Methods for Selecting values of Input Variables in the Analysis of Output from a Computer Code. Technometrics, 221, 239-245

Morris, M.D. (1991): Factorial plans for preliminary computational experiments. Technometrics, 33(2), 161-174

Pettitt, A.N. (1979): A non-parametric Approach to the Change-point Problem. Applied Statistics, 28, 126-135

Saltelli, A., Chan, K., Scott, E.M. (eds.) (2000): Sensitivity Analysis. J. Wiley & Sons, Chichester

Saltelli, A., Ratto, M., Andres, T., Campolongo, F., Cariboni, J., Gatelli, D., Saisana, M., Tarantola, S. (2008): Global Sensitivity Analysis. The Primer. J. Wiley & Sons, Chichester

Saltelli, A., Tarantola, S., Campolongo, F., Ratto, M. (2004): Sensitivity Analysis in Practice: A Guide to Assessing Scientific Models. J. Wiley & Sons, Chichester

Schulzweida, U. , Kornblueh, L., Quast, R. (2004): Climate Data Operators. Max-Planck-Institute for Meteorology,
http://www.mpimet.mpg.de/fileadmin/software/cdo/

Waszkewitz, J., Lenzen, P., Gillet, N. (2001): The PINGO package: Procedural interface for Grib formatted objects. Max-Planck-Institute for Meteorology, Hamburg and
http://www.mad.zmaw.de/Pingo/pingohome.html

Wenzel, V., Kücken, M., Flechsig, M. (1995): MOSES - Modellierung und Simulation ökologischer Systeme. PIK-Report No. 13, Potsdam Institute for Climate Impact Research, Potsdam
http://www.pik-potsdam.de/research/publications/pikreports/summary-report-no-13

Wenzel, V., Matthäus, E., Flechsig, M. (1990): One Decade of SONCHES. Syst. Anal. Mod. & Sim. 7, 411-428

Wierzbicki, A.P. (1984): Models and Sensitivity of Control Systems. Studies in Automation and Control. Vol. 5. Elsevier, Amsterdam

15 Appendices

The appendices summarize the current version implementation, list the examples for model interfaces, user-defined operators and result import interfaces, and they compile all experiment post-processor built-in operators. Finally, a glossary of the main terms as used in this User Guide is supplied.

15.1 Version Implementation

Currently, SimEnv is running under Unix and Linux. For all installations, only the latest version is supported and bug fixes are installed on demand. Tab. 15.1 lists the directory structure of SimEnv. For SimEnv home directories at PIK check Tab. 3.3.

Tab. 15.1 *SimEnv installation directory structure*

Sub-directory of $SE_HOME	Contents
Latest version of SimEnv	
bin	SimEnv scripts, binaries and auxiliary files
lib	SimEnv libraries and scripts to link models and operators
inc	SimEnv include files for models and operators
doc	SimEnv User Guide in pdf
exa	SimEnv examples as used in the User Guide
Version repository of SimEnv at PIK	
version_archive	SimEnv version archive. Version <x.yz> is located in a sub-folder <x.yz> and structured in this sub-folder in the same manner as the latest version.

15.1.1 System Requirements

Tab. 15.2 *System requirements for running SimEnv*

Component	Specification	
	Unix	Linux
hardware	IBM RS6000 and compatibles	Intel-based systems and compatibles with regular 32-bit or 64-bit processors (i386)
operating system	AIX Version 4.3 http://www-03.ibm.com/servers/aix/	any distribution with the Linux kernel
shell	Bourne shell sh	
Python	Version 2.3 http://www.python.org	
NetCDF	Version 3.6.0 http://www.unidata.ucar.edu/packages/netcdf	
Fortran compiler	xlf Version 8.0 IBM Fortran compiler http://www-306.ibm.com/software/awdtools/fortran/xlfortran/	gfortran Version 4.2 GNU Fortran 95 compiler http://gcc.gnu.org/ or Intel ifort Version 10.0 http://www.intel.com/cd/software/products/asmo-na/eng/282048.htm
C/C++ compiler	xlc Version 7.0 IBM C/C++ compiler http://www-306.ibm.com/software/awdtools/xlcpp/ For the compiler the symbolic link “cc” is used.	gcc Version 3.3 GNU C/C++ compiler http://gcc.gnu.org/

	Specification
Java	Version 1.4 http://www.java.com
Matlab (only for running interfaced Matlab models)	Version 7.7 http://www.mathworks.com
Mathematica (only for running interfaced Mathematica models)	Version 4.1 http://wwwwolfram.com
GAMS (only for running interfaced GAMS models)	Distribution 20 http://www.gams.com
MPI (only for performing experiments in compute cluster mode – see Section 7.3)	Version 1.0 http://www.mpi-forum.org
For the visualization framework SimEnvVis	
OpenDX	Version 4.4.4 http://www.opendx.org
Ferret	Version 4.4 http://ferret.wrc.noaa.gov/Ferret/
OpenGL	Version 1.4 http://www.opengl.org/
Qt	Version 3.3.5 http://www.trolltech.com/products/qt

The version number of the software products in the above Tab. 15.2 represent these version, SimEnv was developed with. Higher versions should also be applicable.

For setting up SimEnv, gunzip, tar, configure, make, the xlf and/or gfortran Fortran compiler, and the C/C++ compiler have to be installed. After installing SimEnv, the file $SE_HOME/bin/simenv_settings.txt has to be adapted to the local settings: check Tab. 10.1.

15.1.2 Technical Limitations

Tab. 15.3 *Current SimEnv technical limitations*

Entity		Maximum entity value
Directory strings ($SE_HOME, current workspace; in user-defined files and operators)		
resolved length (relative to absolute paths, environ. variables resolved)	[characters]	128
User-defined files entities (cf. also Section 11.1)		
length of a record in a user-defined file	[characters]	160
length of all general descriptions descr	[characters]	512
length of a local description descr	[characters]	128
length of a unit	[characters]	32
length of a name	[characters]	20
number of user-defined and composed operators in <model>.odf		45
length of all define strings for a macro or a composed operator	[characters]	512
length of a record of a referred ASCII data file	[characters]	1 024

Entity	Maximum entity value
Model interface and experiment preparation entities	
dimensionality of a model output variable	9
dimensionality of a model output variable for Java models	4
number of model output variables	100
number of coordinates	30
number of experiment factors	50
number of slice definitions during interfacing a Fortran/C/C++ model	30
number of single model runs in an experiment	999 999
number of coordinate values and sampled factor values	200 000
storage size of a (float / real*4) model output variable for GAMS models [Mbytes]	64
Experiment post-processing entities	
length of the optional result description string [characters]	128
length of the optional result unit string [characters]	32
number of arguments of an operator	9
dimensionality of a result	9
length of a complete result string (with description and unit) [characters]	512
number of all operands and operators of a result	200
length of a string for a constant [characters]	20
number of constants of a result	30
number of allocatable main memory segments	10
allocatable main memory [MBytes]	2 048
number of post-processor output files	99

15.1.3 Linking User Models and User-Defined Operators

- User models implemented in C/C++ or Fortran have to be linked with the following libraries to interface them to the simulation environment
  - $SE_HOME/lib/libsimenv.a
  - libnetcdf.a from /usr/local/lib or /usr/lib
- User-defined operators to be used in experiment post-processing have to be linked with the following library to interface them to the simulation environment
  - $SE_HOME/libsimenv.a

For running interfaced models outside SimEnv check Section 5.12.

15.1.4 Example Models and User Files

For the following models corresponding files of Tab. 10.7 of can be copied from the corresponding examples-directory $SE_HOME/exa to the user's current workspace by running the SimEnv service command simenv.cpy <model> from this workspace:

Tab. 15.4 *Implemented example models for the current version*
For the generic model "world" check Example 1.1

model	Language / source code	Explanation
world_f	Fortran world_f.f	global atmosphere - biosphere model at resolution of (lat x lon x level x time) = (45 x 90 x 4 x 20)
world_c	C world_c.c	

model	Language / source code	Explanation
world_cpp	C++ world_cpp.cpp	
world_py	Python world_py.py	
world_ja	Java world_ja.java	
world_m	Matlab world_m.m	
world_sh	Shell script level world_sh.f world_shput.f	
world_as	ASCII interface world_as.f	
world_f_auto (semi-automated model interface)	Fortran world_f_auto.f	
world_sh_auto (semi-automated model interface)	Shell script level world_sh.f world_shput.f	
world_f_1x1	Fortran world_f_1x1.f	global atmosphere - biosphere model at a resolution of (lat x lon x level x time) = (180 x 360 x 16 x 20)
world_f_05x05	Fortran world_f_05x05.f	global atmosphere - biosphere model at a resolution of (lat x lon x level x time) = (360 x 720 x 16 x 20)
gridcell_f	Fortran gridcell_f.f	atmosphere - biosphere model for one lat-lon grid cell at a resolution of (level x time) = (4 x 20)
gams_model	GAMS gams_model.gms	GAMS example model

Additionally, the following files are available from the example directory $SE_HOME/exa:

Tab. 15.5 *Implemented model and operator related user files for the current version*
For <opr> see Tab. 15.6 below

File	Explanation
<model>.[f | c | cpp | py | ja | m | gms]	model source code (cf. also example files in Section 15.2)
<model>	model executable compiled and linked from <model>.[f | c | cpp]
world.edf_[2 | 3[a|b|c|d] | 4 | 5 | 6]	experiment description files corresponding to Examples 6.2 to 6.6 to be copied to world_[f | c | cpp | py | ja | m | sh].edf and/or world_f_1x1.edf and world_f_05x05.edf
world.post_[2 | 3c | 4 | 5 | bas | adv]	post-processor input file (complete experiment) for world.edf_[2 | 3c | 4 | 5]: simenv.res world_[f | c | cpp | py | ja | m | sh] [new | append | replace] < world_post.edf_[2 | 3c | 4 | 5] and/or all experiments (selected single run <simenv_run_int>): simenv.res world_[f | c | cpp | py | ja | m | sh] [new | append | replace] <simenv_run_int> < world_post.edf_[bas | adv]
world.dat_[3d | 5 | tab]	data files for world.edf_[3d | 5] and/or world.post_adv

File	Explanation
usr_opr_<opr>.[f | c]	source code for user-defined operator <opr>
usr_opr_<opr>	executable for user-defined operator <opr>
land_sea_mask[<nil> | .f]	executable and source code to derive a coarsed land-sea-mask from the file land_sea_mask.05x05
land_sea_mask.05x05	global ASCII land-sea-mask file with a resolution of 0.5° lat x 0.5° lon
read_result_file[<nil> | .f]	executable and source code for the result file import interface of ASCII and IEEE compliant result output

15.1.5 Example User-Defined Operators

The following user-defined operators are available from the example directory $SE_HOME/exa as source code and executables usr_opr_<opr>. All but operator matmul_c (source file usr_opr_<opr>.c) are implemented in Fortran and available as source files usr_opr_<opr>.f.

Tab. 15.6 *Available user-defined operators*

Operator name <opr>	Operator arguments	Explanation	Example
char_test	char_arg1, char_arg2, arg	character test check usr_opr_char_test.f	`char_test('arg11', 'arg22',bios)`
corr_coeff	arg1, arg2	correlation coefficient R	`corr_coeff(bios, -bios) = -1.`
div	arg1, arg2	division as an example how the corresponding built in basic operator works	`div(-2,-4) = 0.5`
matmul_[f | c]	arg1, arg2	matrix multiplication of 2-dimensional operands	`matmul_[f | c] (mat1,mat2)`
simple_div	arg1, arg2	division without consideration of overflow, underflow, and division by 0.	`simple_div(-2,-4) = 0.5`

15.2 Examples for Model Interfaces

15.2.1 Example Implementation of the Generic Model world

According to Example 1.1 on page 6 dynamics of the model world depend on four model paramters p1, p2, p3, and p4:

Tab. 15.7 *Factors of the generic model world*
*Mapping between model factors and internal model parameters is performed by the model coupling interface functions simenv_get_**

Model factor	Factor default value	Internal model parameter name	Factor unit	Factor meaning
p1	1.	phi_lat	π/12	latitudinal phase shift
p2	2.	omega_lat	2*π	latitudinal frequency
p3	3.	phi_lon	π/12	longitudinal phase shift
p4	4.	omega_lon	2*π	longitudinal frequency

For reasons of simplification these factors (parameters) influence state variables atmo and bios by the product of two trigonometric terms value_lat and value_lon in the following manner:

```
value_lat(lat)           = sin( 2*π*omega_lat * f(lat) + phi_lat*π/12 )
value_lon(lon)           = sin( 2*π*omega_lon * f(lon) + phi_lon*π/12 )
```

The function f(.) norms value_lat and value_lon by lat and/or lon in a way, that it holds

```
value_[lat|lon](1)       = sin( +π*omega_[lat|lon] + phi_[lat|lon]*π/12 )
value_[lat|lon](last/2)  = sin( ±0*omega_[lat|lon] + phi_[lat|lon]*π/12 )
value_[lat|lon](last)    = sin( -π*omega_[lat|lon] + phi_[lat|lon]*π/12 )
```

Finally,

```
atmo(lat,lon,level,time) = value_lat(lat) * value_lon(lon) * (100*time+level-1)
bios(lat,lon,time)       = value_lat(lat) * value_lon(lon) *  100*time
```

and - notated in the syntax of the SimEnv post-processor -

```
atmo_g(time)             = avg_l('001',abs(atmo(lat,lon,1,time)))
bios_g                   = avg(abs(bios(lat,lon,time)))
```

Means avg and avg_l are calculated in a box with the extent Δlat x Δlon = 10° x 10° and (lat,lon) = (0°,0°) in the mid of the box.

15.2.2 Fortran Model

With respect to Example 5.1 the following Fortran code **world_f.f** could be used to describe the model interfaced to SimEnv. SimEnv modifications are marked in **bold**.

```
      program world_f
      c  declare SimEnv interface functions (compile with -I$SE_HOME/inc)
         include 'simenv_mod_f.inc'
         integer*4 simenv_sts,simenv_run_int
         character*6 simenv_run_char
      c  declare atmo without dimensions level and time and bios without time
      c  because they are computed in place and simenv_slice_f is used
         real*4      atmo(0:44,0:89)
         real*4      bios(0:35,0:89)
         integer*4   atmo_g(0:19)
         integer*4   bios_g

         p1 = 1.
         p2 = 2.
         p3 = 3.
         p4 = 4.

         simenv_sts = simenv_ini_f()
      c  check return code for the model interface functions at least here
         if(simenv_sts.ne.0) stop 1
      c  simenv_get_run_f only if necessary:
         simenv_sts = simenv_get_run_f(simenv_run_int,simenv_run_char)
         simenv_sts = simenv_get_f('p1',p1,p1)
         simenv_sts = simenv_get_f('p2',p2,p2)
         simenv_sts = simenv_get_f('p3',p3,p3)
         simenv_sts = simenv_get_f('p4',p4,p4)

      c  compute dynamics of atmo and bios over space and time,
      c  of atmo_g over time, all dependent on p1,p2,p3,p4
         do idecade = 0,19
         ...
            do level= 0,3
               simenv_sts = simenv_slice_f('atmo',3,level+1,level+1)
               simenv_sts = simenv_slice_f('atmo',4,idecade+1,idecade+1)
               simenv_sts = simenv_put_f('atmo',atmo)
            enddo
            simenv_sts = simenv_slice_f('bios',3,idecade+1,idecade+1)
            simenv_sts = simenv_put_f('bios',bios)
         enddo
         ...
         simenv_sts = simenv_put_f('atmo_g',atmo_g)
      c  compute dynamics of bios_g
         ...
         simenv_sts = simenv_put_f('bios_g',bios_g)

         simenv_sts = simenv_end_f()
         end
```

Example file: world_f.f

Example 15.1 *Model interface for Fortran models – model world_f.f*

15.2.3 Fortran Model with Semi-Automated Model Interface

With respect to Example 5.1 the following Fortran code **world_f_auto.f** could be used to describe the model interfaced semi-automatedly to SimEnv. SimEnv modifications are marked in **bold**.

```
      program world_f_auto
      c  declare SimEnv interface functions (compile with -I$SE_HOME/inc)
      c  simenv_sts, simenv_run_int and simenv_run_char are also declared
         include 'simenv_mod_auto_f.inc'
      c  declare atmo without dimensions level and time and bios without time
      c  because they are computed in place and simenv_slice_f is used
         real*4      atmo(0:44,0:89)
         real*4      bios(0:35,0:89)
         integer*4   atmo_g(0:19)
         integer*4   bios_g

         p1 = 1.
         p2 = 2.
         p3 = 3.
         p4 = 4.

      c  include source code sequence for the semi-automated model interface
         include 'world_f_auto_f.inc'

      c  compute dynamics of atmo and bios over space and time,
      c  of atmo_g over time, all dependent on p1,p2,p3,p4
         do idecade = 0,19
         ...
            do level= 0,3
               simenv_sts = simenv_slice_f('atmo',3,level+1,level+1)
               simenv_sts = simenv_slice_f('atmo',4,idecade+1,idecade+1)
               simenv_sts = simenv_put_f('atmo',atmo)
            enddo
            simenv_sts = simenv_slice_f('bios',3,idecade+1,idecade+1)
            simenv_sts = simenv_put_f('bios',bios)
         enddo
         ...
         simenv_sts = simenv_put_f('atmo_g',atmo_g)
      c  compute dynamics of bios_g
         ...
         simenv_sts = simenv_put_f('bios_g',bios_g)

         simenv_sts = simenv_end_f()
         end
```

Example file: world_f_auto.f

Example 15.2 *Semi-automated model interface for Fortran models – model world_f_auto.f*

15.2.4 C Model

With respect to Example 5.1 the following C code **world_c.c** could be used to describe the model interfaced to SimEnv. SimEnv modifications are marked in **bold**.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* declare SimEnv interface functions (compile with -I$SE_HOME/inc)
#include "simenv_mod_c.inc"

/* declare atmo without dimensions level and time and bios without time*/
/* because they are computed in place and simenv_slice_c is used */
static float   atmo[45][90];
static float   bios[36][90];
static int     atmo_g[20];
static int     bios_g;

main(void)
{

   float p1,p2,p3,p4;
   int level,idecade,simenv_sts,simenv_run_int,level1,idecade1,idim;
   char simenv_run_char[6];
   p1 = 1.;
   p2 = 2.;
   p3 = 3.;
   p4 = 4.;

   simenv_sts = simenv_ini_c();
/* check return code of model interface functions at least here */
   if(simenv_sts != 0) return 1;
/* simenv_get_run_c only if necessary: */
   simenv_sts = simenv_get_run_c(&simenv_run_int,simenv_run_char);
   simenv_sts = simenv_get_c("p1",&p1,&p1);
   simenv_sts = simenv_get_c("p2",&p2,&p2);
   simenv_sts = simenv_get_c("p3",&p3,&p3);
   simenv_sts = simenv_get_c("p4",&p4,&p4);
/* compute dynamics of atmo and bios over space and time, */
/* of atmo_g over time, all dependent on p1,p2,p3,p4 */
   for (idecade=0; idecade<=19; idecade++)
   {...
     for (level=0; level<=3; level++)
     { ...
       idim=3;
       level1=level+1;
       simenv_sts = simenv_slice_c("atmo",&idim,&level1,&level1);
       idim=4;
       idecade1=idecade+1;
       simenv_sts = simenv_slice_c("atmo",&idim,&idecade1,&idecade1);
       simenv_sts = simenv_put_c("atmo",(char *) &atmo);
     }
     idim=3;
     idecade1=idecade+1;
     simenv_sts = simenv_slice_c("bios",&idim,&idecade1,&idecade1);
     simenv_sts = simenv_put_c("bios",(char *) &bios);
   }
```

```
      simenv_sts = simenv_put_c("atmo_g",(char *) &atmo_g);

   /* compute dynamics of bios_g */
      ...
      simenv_sts = simenv_put_c("bios_g", ,(char *) &bios_g);
      simenv_sts = simenv_end_c();
      return 0;
   }
```

Example file: world_c.c

Example 15.3 *Model interface for C models – model world_c.c*

15.2.5 C++ Model

With respect to Example 5.1 the following C++ code **world_cpp.cpp** could be used to describe the model interfaced to SimEnv. SimEnv modifications are marked in **bold**.

```
#include <stdio.h>
#include <stdlib.h>
/* declare SimEnv interface functions (compile with -I$SE_HOME/inc)
#include "simenv_mod_c.inc"

class World
{
/* declare atmo without dimensions level and time and bios without time*/
/* because they are computed in place and simenv_slice_c is used */
   public:  float   atmo[45][90];
   public:  float   bios[36][90];
   public:  int     atmo_g[20];
   public:  int     bios_g;
   private: int     level,idecade,simenv_sts,level1,idecade1,idim;

   public: void computeAtmo(float p1 ,float p2, float p3, float p4)
/* compute dynamics of atmo over space and time, */
/* and of atmo_g over time, all dependent on p1,p2,p3,p4 */
   {
     for (idecade=0; idecade<=19; idecade++)
     {...
       for (level=0; level<=3; level++)
       {...
         idim=3;
         level1=level1+1;
         simenv_sts = simenv_slice_c("atmo",&idim,&level,&level);
         idim=4;
         idecade1=idecade1+1;
         simenv_sts = simenv_slice_c("atmo",&idim,&idecade,&idecade);
         simenv_sts = simenv_put_c("atmo",(char *) &atmo);
       }
     }
   }

public: void computeBios(float p1, float p2, float p3, float p4)
/* compute dynamics of bios over space and time, */
/* and of bios_g all dependent on p1,p2,p3,p4 */
   {
     int simenv_sts;
     for (idecade=0; idecade<=19; idecade++)
     {...
       idim=3;
       idecade1=idecade1+1;
       simenv_sts = simenv_slice_c("bios",&idim,&idecade1,&idecade1);
       simenv_sts = simenv_put_c("bios",(char *) &bios);
     }
/* compute dynamics of bios_g */
     ...
   }
}
```

```
main(void)
{
   int simenv_sts,simenv_run_int;
   char simenv_run_char[6];
   float p1 = 1.;
   float p2 = 2.;
   float p3 = 3.;
   float p4 = 4.;

   simenv_sts = simenv_ini_c();
/* check return code of model interface functions at least here */
   if(simenv_sts != 0) return 1;
/* simenv_get_run_c only if necessary: */
   simenv_sts = simenv_get_run_c(&simenv_run_int,simenv_run_char);

   simenv_sts = simenv_get_c("p1",&p1,&p1);
   simenv_sts = simenv_get_c("p2",&p2,&p2);
   simenv_sts = simenv_get_c("p3",&p3,&p3);
   simenv_sts = simenv_get_c("p4",&p4,&p4);

   World world;
   world.computeAtmo(p1,p2,p3,p4);
   simenv_sts = simenv_put_c("atmo_g",(char *) &(world.atmo_g));
   world.computeBios(p1,p2,p3,p4);
   simenv_sts = simenv_put_c("bios_g",(char *) &(world.bios_g));

   simenv_sts = simenv_end_c();
   return 0;
}
```

Example file: world_cpp.cpp

Example 15.4 *Model interface for C++ models – model world_cpp.cpp*

15.2.6 Python Model

With respect to Example 5.1 the following Python code **world_py.py** could be used to describe the model interfaced to SimEnv. SimEnv modifications are marked in **bold**.

```
#!/usr/local/bin/python
import string
import os
from simenv import *
# this model needs the math and the Numeric package
# set the PYTHONPATH environment variable accordingly
from math import *
from Numeric import *

atmo=zeros([45,90,4,20], Float)
bios=zeros([36,90,20], Float)
atmo_g=zeros([20], Float)
p1=1.
p2=2.
p3=3.
p4=4.

simenv_ini_py()
# simenv_get_run_py only if necessary:
simenv_run_int = int(simenv_get_run_py())

p1 = float(simenv_get_py('p1',p1))
p2 = float(simenv_get_py('p2',p2))
p3 = float(simenv_get_py('p3',p3))
p4 = float(simenv_get_py('p4',p4))

# compute dynamics of atmo and bios over space and time,
# of atmo_g over time, all dependent on p1,p2,p3,p4
for idecade in range(20):
   ...
   for level in range(4):
      ...
atmo=reshape(atmo,45*90*4*20,))
simenv_put_py('atmo',atmo)
bios=reshape(atmo,45*90*20,))
simenv_put_py('bios',bios)
simenv_put_py('atmo_g',atmo_g)
# compute dynamics of bios_g
# ...
simenv_put_py('bios_g',bios_g)
simenv_end_py()
```

Example file: world_py.py

Example 15.5 *Model interface for Python models – model world_py.py*

15.2.7 Java Model

With respect to Example 5.1 the following Java code **world_ja.java** could be used to describe the model interfaced to SimEnv. SimEnv modifications are marked in **bold**.

```
import java.io.*;

public class world_ja {
   static float[][][][]  atmo =   new float[45][90][4][20];
   static float[][][]    bios =   new float[36][90][20];
   static int[]          atmo_g = new int[20];
   static int            bios_g;

   public static void main(String[] args) {
       int      simenv_sts, simenv_run_int;
       int      idecade, level;

       float    phi_lat=1.F;
       float  omega_lat=2.F;
       float    phi_lon=3.F;
       float  omega_lon=4.F;

       simenv_sts = Simenv.simenv_ini_ja();
// simenv get run ja only if necessary:
       simenv_run_int = Integer.parseInt(Simenv.simenv_get_run_ja());

       phi_lat = Simenv.simenv_get_ja("p1", phi_lat);
       omega_lat = Simenv.simenv_get_ja("p2", omega_lat);
       phi_lon = Simenv.simenv_get_ja("p3", phi_lon);
       omega_lon = Simenv.simenv_get_ja("p4", omega_lon);

// compute dynamics of atmo and bios over space and time,
// of atmo_g over time, all dependent on p1,p2,p3,p4
       for (idecade=0; idecade<20; idecade++)
           ...
           for (level=0; level<4; level++)
           ...
       simenv_sts = Simenv.simenv_put_ja("atmo", atmo);
       simenv_sts = Simenv.simenv_put_ja("bios", bios);
       simenv_sts = Simenv.simenv_put_ja("atmo_g", atmo_g);
       simenv_sts = Simenv.simenv_put_ja("bios_g", bios_g);
           ...
       simenv_sts=Simenv.simenv_end_ja();

       System.exit(0);

       }
   }
```

Example file: world_ja.java

Example 15.6 *Model interface for Java models – model world_ja.java*

15.2.8 Matlab Model

With respect to Example 5.1 the following Matlab code **world_m.m** could be used to describe the model interfaced to SimEnv. SimEnv modifications are marked in **bold**.

```
atmo = zeros(45,90,4,20);
bios = zeros(36,90,20);
atmo_g = zeros(20);

phi lat=1.;
omega_lat=2.;
phi lon=3.;
omega lon=4.;

simenv sts = simenv ini m();
% simenv get run m only if necessary:
simenv_run_int = str2num(simenv_get_run_m());

phi lat = simenv get m('p1', phi lat);
omega_lat = simenv_get_m('p2', omega_lat);
phi lon = simenv get m('p3', phi lon);
omega lon = simenv get m('p4', omega lon);

% compute dynamics of atmo and bios over space and time,
% of atmo_g over time, all dependent on p1,p2,p3,p4
for idecade = 0:19
        ...
        for level = 0:3
        ...
        end
end
simenv_sts = simenv_put_m('atmo', single(atmo));
simenv sts = simenv put m('bios', single(bios));
simenv sts = simenv put m('atmo g', int32(atmo g));
simenv sts = simenv put m('bios g', int32(bios g));
        ...
simenv end m();
```

Example file: world_m.m

Example 15.7 *Model interface for Matlab models – model world_m.m*

15.2.9 Mathematica Model

Example 15.8 describes the model interface for a Mathematica model. The model itself is not provided.

```
#! /bin/sh

# perform always and as the first $SE_HOME/bin/simenv_*_sh dot script

# get current run number <simenv_run_char> and <simenv_run_int>
. $SE_HOME/bin/simenv_get_run_sh

# get factor names and adjusted values and
# run the Mathematica model <model>.m
. $SE_HOME/bin/simenv_run_mathematica

# transfer ASCII model output files to SimEnv model output
# (cf. Example 15.10 and Example 15.11)
# ...

# remove temporary sub-directory run$simenv_run_char
rmdir run$simenv_run_char

# perform always and as the last $SE_HOME/bin/simenv_*_sh dot script
. $SE_HOME/bin/simenv_end_sh
```

Example 15.8 *Model interface for Mathematica – model shell script <model>.run*

15.2.10 GAMS Model

SimEnv comes with an interfaced GAMS model **gams_model.gms** and all associated files that fully correspond to the GAMS example model at http://www.gams.com/docs/gams/Tutorial.pdf. Modifications for SimEnv are marked in **bold**.

```
SETS
   I     canning plants   / SEATTLE, SAN-DIEGO /
   J     markets          / NEW-YORK, CHICAGO, TOPEKA / ;

PARAMETERS
   A(I)  capacity of plant i in cases
     /    SEATTLE     350
          SAN-DIEGO   600  /
   B(J)  demand at market j in cases
     /    NEW-YORK    325
          CHICAGO     300
          TOPEKA      275  / ;

* - Before using parameters (here dem_ny and dem_ch) as SimEnv experiment
*   factors they have to be declared as GAMS model parameters
*   with default values from above.
* - Then insert $include <model>_simenv_get.inc
*   simenv_get.inc is generated automatically based on <model>.edf
* - And assign adjusted factors to model variables
   PARAMETERS
   dem_ny /325.0/;
   dem_ch /300.0/;
   $include gams_model_simenv_get.inc
   A("SEATTLE")   = dem_ny;
   A("SAN-DIEGO") = dem_ch;

TABLE  D(I,J)  distance in thousands of miles
                 NEW-YORK       CHICAGO      TOPEKA
   SEATTLE          2.5           1.7          1.8
   SAN-DIEGO        2.5           1.8          1.4 ;
SCALAR F  freight in dollars per case per thousand miles  /90/

* get the model status as a model output
   modstat is set to transport.modelstat ;

PARAMETER C(I,J)  transport cost in thousands of dollars per case ;
   C(I,J) = F * D(I,J) / 1000 ;
VARIABLES
   X(I,J)  shipment quantities in cases
   Z       total transportation costs in thousands of dollars ;
POSITIVE VARIABLE X ;
EQUATIONS
   COST        define objective function
   SUPPLY(I)   observe supply limit at plant i
   DEMAND(J)   satisfy demand at market j ;
COST ..        Z  =E=  SUM((I,J), C(I,J)*X(I,J)) ;
SUPPLY(I) ..   SUM(J, X(I,J))  =L=  A(I) ;
DEMAND(J) ..   SUM(I, X(I,J))  =G=  B(J) ;
MODEL TRANSPORT /ALL/ ;
SOLVE TRANSPORT USING LP MINIMIZING Z ;
```

```
* After solving the equations $include simenv_put.inc
* has to be inserted.
* simenv_put.inc is generated automatically by SimEnv
* based on <model>.edf and <model>.gdf
* Additional GAMS statements are possible after the $include statement
   modstat = transport.modelstat
   $include gams_model_simenv_put.inc

* Only if sub-models sub_m1 and sub_m2 are coupled (cf. Example 5.3):
* $call "gams ../sub_m1.gms ll= lo=2 lf=sub_m1.nlog dp=0 Optdir=../";
* $call "gams ../sub_m2.gms ll= lo=2 lf=sub_m2.nlog dp=0 Optdir=../";
```

Example file: gams_model.gms

Example 15.9 *Model interface for GAMS models – model gams_model.gms*

15.2.11 Model Interface at Shell Script Level

Assume any experiment. Assume model executable world_sh to take factor values p1 to p4 as arguments from the command line.
The shell script **world_sh.run** with an interface at shell script level to run the model world_sh and to transform model output to SimEnv could look like:

```
#! /bin/sh

p1=1.
p2=2.
p3=3.
p4=4.

# perform always and as the first $SE_HOME/bin/simenv_*_sh dot script
# altern. perform . $SE_WS/<model>_sh.inc for semi-autom. model interface
. $SE_HOME/bin/simenv_ini_sh

# get current run number <simenv_run_char> and <simenv_run_int>
. $SE HOME/bin/simenv get run sh

# get adjusted values for factors p1 ... p4
factor_name='p1'
factor_def_val=$p1
. $SE_HOME/bin/simenv_get_sh
factor_name='p2'
factor_def_val=$p2
. $SE_HOME/bin/simenv_get_sh
factor_name='p3'
factor_def_val=$p3
. $SE_HOME/bin/simenv_get_sh
factor_name='p4'
factor_def_val=$p4
. $SE_HOME/bin/simenv_get_sh

# create temporary directory run<simenv_run_char> to perform the model
# and model output transformation from native to SimEnv structure there
mkdir run$simenv_run_char
cd run$simenv_run_char

# run the model
cp ../land_sea_mask.coarsed .
../world_sh $p1 $p2 $p3 $p4

# read model results and output them to SimEnv
../world_shput

# clear and remove directory
cd ..
rm -fR run$simenv_run_char

# perform always and as the last $SE_HOME/bin/simenv_*_sh dot script
. $SE_HOME/bin/simenv_end_sh
```

Example file: world_sh.run

Example 15.10 *Model interface at shell script level – model shell script world_sh.run*

15.2.12 Model Interface for ASCII Files

Assume any experiment. Assume model executable world_as (example file world_as.f)

- to take factor names and resulting adjusted values p1 to p4 from the file generated by simenv_get_as
- to output model variables to ASCII files
  - atmo_bios.ascii<simenv_run_char>
  - atmo_g.ascii<simenv_run_char>
  - bios_g.ascii<simenv_run_char>

  with the same file structure as in Example 5.6. The current run number in 6-character notation is appended to the file names to distinguish files for parallel experiment performance.

The shell script **world_as.run** with an ASCII interface to run the model world_as and to transfer model output to SimEnv could look like:

```
#! /bin/sh

# perform always and as the first $SE_HOME/bin/simenv_*_sh dot script
# altern. perform . $SE_WS/<model>_sh.inc for semi-autom. model interface
. $SE_HOME/bin/simenv_ini_sh

# get current run number <simenv_run_char> and <simenv_run_int>
. $SE_HOME/bin/simenv_get_run_sh

# get factor names and adjusted values
# to ASCII file world_as.as<simenv_run_char>
. $SE_HOME/bin/simenv_get_as

# run the model:
# read world_as.as$simenv_run_char
# store model output to ASCII files
./world_as

# transfer ASCII model output files to SimEnv model output
# use simenv_put_as_simple since the ASCII file has 9000 columns:
$SE_HOME/bin/simenv_put_as_simple atmo_bios.ascii$simenv_run_char lat
# use simenv_put_as since the ASCII files have 1 column:
$SE_HOME/bin/simenv_put_as atmo_g.ascii$simenv_run_char time
$SE_HOME/bin/simenv_put_as bios_g.ascii$simenv_run_char

# remove ASCII files
rm -f world_as.as$simenv_run_char
rm -f atmo_bios.ascii$simenv_run_char
rm -f atmo_g.ascii$simenv_run_char
rm -f bios_g.ascii$simenv_run_char

# perform always and as the last  $SE_HOME/bin/simenv_*_sh dot script
. $SE_HOME/bin/simenv_end_sh
```

Example file: world_as.run

Example 15.11 *Model interface for ASCII files – model shell script world_as.run*

15.2.13 Semi-Automated Model Interface at Shell Script Level

Assume any experiment. Assume model executable world_sh to take factor values p1 to p4 as arguments from the command line.
The shell script **world_sh_auto.run** with a semi-automated interface at shell script level to run the model world_sh and to transform model output to SimEnv could look like:

```
#! /bin/sh

p1=1.
p2=2.
p3=3.
p4=4.

# perform dot script world_sh_auto_sh.inc
# for semi-automated model interface at shell script level
# alternatively perform dot script $SE_HOME/bin/simenv_ini_sh
. $SE_WS/world_sh_auto_sh.inc

# create temporary directory run<simenv run char> to perform the model
# and model output transformation from native to SimEnv structure there
mkdir run$simenv_run_char
cd run$simenv_run_char

# run the model
cp ../land_sea_mask.coarsed .
../world_sh $p1 $p2 $p3 $p4

# read model results and output them to SimEnv
../world_shput

# clear and remove directory
cd ..
rm -fR run$simenv_run_char

# perform always and as the last $SE_HOME/bin/simenv_*_sh dot script
. $SE_HOME/bin/simenv_end_sh
```

Example file: world_sh_auto.run

Example 15.12 *Semi-automated model interface at shell script level – model shell script world_sh_auto.run*

15.3 Example Implementation for the Experiment Post-Processor User-Defined Operator matmul_[f | c]

15.3.1 Fortran Implementation

Implementation of the user-defined operator matmul_f in the file **usr_opr_matmul_f.f**:

```
      integer*4 function simenv_check_user_def_operator()
c     declare SimEnv interface functions (compile with -I$SE_HOME/inc)
      include 'simenv_opr_f.inc'
c     declare fields to hold extents and coordinates
      dimension iext1(9),iext2(9)
      dimension ico_nr1(9),ico_nr2(9)
      dimension ico_beg_pos1(9),ico_beg_pos2(9)
      character*20 co_name1(9),co_name2(9)

c     get dimensionality idimens, extents iext,
c     formal coordinate number ico_nr and
c     formal coordinate begin position ico_beg_pos
      idimens1=simenv_get_dim_arg_f(1,iext1)
      idimens2=simenv_get_dim_arg_f(2,iext2)
      iok=simenv_get_co_arg_f(1,ico_nr1,ico_beg_pos1,co_name1)
      iok=simenv_get_co_arg_f(2,ico_nr2,ico_beg_pos2,co_name2)
c     get check modus for coordinates
      ichk_modus=simenv_get_co_chk_modus_f()

      if(idimens1.ne.2.or.idimens2.ne.2) then
c     wrong dimensionalities
         ierror=1
      else
         if(iext1(2).ne.iext2(1)) then
c     wrong extents
            ierror=2
         else
            if(ico_nr1(2).eq.ico_nr2(1)) then
c     coordinates identical
               if(ico_beg_pos1(2).eq.ico_beg_pos2(1)) then
                  iret=31
               else
                  iret=33
               endif
            else
c     differing coordinates
               iret=32
               if(ichk_modus.eq.1) then
c     check only for weak coordinate
                  do j=0,iext1(2)-1
c     get coordinate values
                     iretv1=simenv_get_co_val_f(
     #                    ico_nr1(2),ico_beg_pos1(2)+j,value1)
                     iretv2=simenv_get_co_val_f(
     #                    ico_nr2(1),ico_beg_pos2(1)+j,value2)
c     iret=33: differing coordinate values
```

```
                  if(value1.ne.value2) iret=33
               enddo
            endif
         endif

         ierror=0
         if(ichk_modus.eq.2) then
            if(iret.gt.31) ierror=3
         elseif(ichk_modus.eq.1) then
            if(iret.gt.32) ierror=3
         endif

      endif
   endif

if(ierror.eq.0) then
     iext1(2)=iext2(2)
     ico_nr1(2)=ico_nr2(2)
     ico_beg_pos1(2)=ico_beg_pos2(2)
     iok=simenv_put_struct_res_f(0,idimens1,iext1,ico_nr1,ico_beg_pos1)
   endif

c  return error code
   simenv_check_user_def_operator=ierror
   return
   end


   integer*4 function simenv_compute_user_def_operator(res)
c  SimEnv operator results are always of type real*4
   real*4 res(1)
c  declare SimEnv interface functions (compile with -I$SE_HOME/inc)
   include 'simenv_opr_f.inc'
c  auxiliary variables
   integer*4 iext1(9),iext2(9)
   real*8 value8,r8fac1,r8fac2

c  get dimensionality idimens and extents iext for both arguments
   idimens=simenv_get_dim_arg_f(1,iext1)
   idimens=simenv_get_dim_arg_f(2,iext2)

c  perform matrix multiplication
   m=0
   do k=1,iext2(2)
      iarg2_offs=(k-1)*iext2(1)
      do i=1,iext1(1)
         iarg1_offs=i
c   res(i,k) = sum(arg1(i,l) * arg2(l,k))
         value8=0.
         indi_defined=0
         do l=1,iext1(2)
            ia1=iarg1_offs+(l-1)*iext1(1)
            ia2=iarg2_offs+l
            fac1=simenv_get_arg_f(1,ia1)
            fac2=simenv_get_arg_f(2,ia2)
            if(simenv_chk_undef_f(fac1)+simenv_chk_undef_f(fac2).eq.0)
            then
```

```
                  indi_defined=1
                  r8fac1=fac1
                  r8fac2=fac2
                  value8=value8+r8fac1*r8fac2
               endif
            enddo
            m=m+1
            if(indi_defined.eq.0) then
               res(m)=simenv_put_undef_f()
            else
               res(m)=simenv_clip_undef_f(value8)
            endif
         enddo
      enddo

c     return error code
      simenv_compute_user_def_operator=0
      return
      end
```

Example file: usr_opr_matmul_f.f

Example 15.13 *Experiment post-processor user-defined operator module – operator matmul_f*

15.3.2 C Implementation

Implementation of the user-defined operator matmul_c in the file **usr_opr_matmul_c.c**:

```
#include <strings.h>
#include <stdio.h>
#include "simenv_opr_c.inc"          /* compile with -I$SE_HOME/inc */

int simenv_check_user_def_operator()
{
   int iext1[9],iext2[9];
   int ico_nr1[9],ico_nr2[9],ico_beg_pos1[9],ico_beg_pos2[9];
   char co_name1[180],co_name2[180];
   int idimens1, idimens2;
   int ichk_modus;
   int iret,iretv1,iretv2,j,iok,ierror=0;
   float value1, value2;

/* get dimensionality idimens, extents iext,
   formal coordinate number ico_nr and
   formal coordinate begin position ico_beg_pos
*/
   idimens1=simenv_get_dim_arg_c(1,iext1);
   idimens2=simenv_get_dim_arg_c(2,iext2);
   iok=simenv_get_co_arg_c(1,ico_nr1,ico_beg_pos1,co_name1);
   iok=simenv_get_co_arg_c(2,ico_nr2,ico_beg_pos2,co_name2);

   ichk_modus=simenv_get_co_chk_modus_c();

   if(idimens1!=2 || idimens2!=2)
      ierror=1;                      /* wrong dimensionalities */
   else
      if(iext1[1]!=iext2[0])
         ierror=2;                   /* wrong dimensions */
      else
         { if(ico_nr1[1]==ico_nr2[0])
              if(ico_beg_pos1[1]==ico_beg_pos2[0])
                 iret=31;
              else
                 iret=33;            /* coordinates identical*/
           else
             { iret=32;              /* differing coordinates */
               if(ichk_modus==1)
                  for (j=0;j<iext1[1];j++) /* only for weak c. check */
                     { /* get coordinate values */
                       iretv1=simenv_get_co_val_c
                              (ico_nr1[1],ico_beg_pos1[1]+j,&value1);
                       iretv2=simenv_get_co_val_c
                              (ico_nr2[0],ico_beg_pos2[0]+j,&value2);
/* iret=33: differing coordinate values */
                              if(value1 != value2)
                                 iret=33;
                     }
             }
```

```
            ierror=0;
            if(ichk_modus==2)
               if(iret>31) ierror=3;
            else
               if(ichk_modus==1)
                  if(iret>32) ierror=3;
         }

   if(ierror==0)
         { iext1[1]=iext2[1];
           ico_nr1[1]=ico_nr2[1];
           ico_beg_pos1[1]=ico_beg_pos2[1];
   iok=simenv_put_struct_res_c(0,idimens1,iext1,ico_nr1,
                                       ico_beg_pos1);
         }
      return ierror; /* return error code */
    }



   /* SimEnv operator results are always of type real*4 */
   int simenv_compute_user_def_operator(float *res)
   {
      int iext1[9],iext2[9];
      double value8,r8fac1,r8fac2;
      int idimens;
      int i,k,l,m,ia1,ia2;
      int iarg1_offs,iarg2_offs,indi_defined;
      float fac1,fac2;

   /* get dimensionality idimens and dimensions idim for both arguments */
      idimens=simenv_get_dim_arg_c(1,iext1);
      idimens=simenv_get_dim_arg_c(2,iext2);

   /* perform matrix multiplication */
      m=0;
      for (k=1;k<=iext2[1];k++)
         { iarg2_offs=(k-1)*iext2[0];
           for (i=1;i<=iext1[0];i++)
              { iarg1_offs=i;
   /* res(i,k) = sum(arg1(i,l) * arg2(l,k)) */
                value8=0.;
                indi_defined=0;
                for (l=1;l<=iext1[1];l++)
                   { ia1=iarg1_offs+(l-1)*iext1[0];
                     ia2=iarg2_offs+l;
                     fac1=simenv_get_arg_c(1,ia1);
                     fac2=simenv_get_arg_c(2,ia2);
                     if(simenv_chk_undef_c(fac1) +
                        simenv_chk_undef_c(fac2)==0)
                        { indi_defined=1;
                          r8fac1=fac1;
                          r8fac2=fac2;
                          value8=value8+r8fac1*r8fac2;
                        }
                   }
                m=m+1;
```

```
                  if(indi_defined==0)
                     res[m-1]=simenv_put_undef_c();
                  else
                     res[m-1]=simenv_clip_undef_c(value8);
               }
            }
         return 0;
      }
```

Example file: usr_opr_matmul_c.c

Example 15.14 *Experiment post-processor user-defined operator module – operator matmul_c*

15.4 Example for an Experiment Post-Processor Result Import Interface

In Example 15.15 an implementation of an interface to import ASCII post-processor output from SimEnv can be found. A corresponding interface to import IEEE compliant post-processor output is documented.

```
      subroutine read_result_file_ascii(model_name,res_nmb)
      character model_name*20,res_nmb*2
      real*4, pointer, dimension(:) :: coord_values
      real*4, pointer, dimension(:) :: result_values
      integer*4 idim, iext(9)
      character result_expr*512, result_desc*128, result_unit*32
      character coord_name*20
      open(unit=1,file=trim(model_name)//'inf'//res_nmb//'.ascii',
     #     form='formatted',status='old')
      open(unit=2,file=trim(model_name)//'res'//res_nmb//'.ascii',
     #     form='formatted',status='old')
      iostat=0
      do while (iostat.eq.0)
         read(1,'(a512)',iostat=iostat) result_expr
         if(iostat.eq.0) then
            read(1,'(a128)',iostat=iostat1) result_desc
            read(1,'(a32)',iostat=iostat1) result_unit
            read(1,'(10i8)',iostat=iostat1) idim,(iext(i),i=1,9)
            length_result=1
            do i=1,idim
               length_result=length_result*iext(i)
               read(1,'(a20)',iostat=iostat1) coord_name
               allocate(coord_values(iext(i)))
               ibeg=1
               do while (ibeg.le.iext(i))
                  iend=min0(ibeg+9,iext(i))
                  read(1,'(10g15.7)',iostat=iostat1) (coord_values(j),
                  ibeg=iend+1                         j=ibeg,iend)
               enddo
c              further processing of coordinate values
c              ...
               deallocate (coord_values)
            enddo
            allocate(result_values(length_result))
            ibeg=1
            do while (ibeg.le.length_result)
               iend=min0(ibeg+9,length_result)
               read(2,'(10g15.7)',iostat=iostat) (result_values(j),
               ibeg=iend+1                        j=ibeg,iend)
            enddo
c           further processing of result values
c           ...
            deallocate(result_values)
         endif
      enddo
      close(unit=1)
      close(unit=2)
      return
      end
```

Example file: read_result_file.f (together with subroutine read_result_file_ieee)

Example 15.15 *ASCII compliant experiment post-processor result import interface*

15.5 List of Experiment Post-Processor Built-In Operators and Operator Arguments

15.5.1 Experiment Post-Processor Built-In Operators (in Thematic Order)

arg	general numerical argument
int_arg	integer constant argument ≥ 0
real_arg	real (float) constant argument
char_arg	character argument

Tab. 15.8 *Experiment post-processor built-in operators (in thematic order)*

Name	Meaning	See
Elemental operators		**Tab. 8.3 on page 86**
arg1 + arg2	addition	
arg1 - arg2	subtraction	
arg1 * arg2	multiplication	
arg1 / arg2	division	
arg1 ** arg2	exponentiation	
+ arg	identity	
- arg	negation	
(arg)	parentheses	
Basic operators		**Tab. 8.4 on page 87**
abs(arg)	absolute value	
dim(arg1,arg2)	positive difference	
exp(arg)	exponential function	
int(arg)	integer truncation value	
log(arg)	natural logarithm	
log10(arg)	decade logarithm	
mod(arg1,arg2)	remainder	
nint(arg)	nearest integer value	
sign(arg)	sign of value	
sqrt(arg)	square root	
Trigonometric operators		**Tab. 8.4 on page 87**
sin(arg)	sine	
cos(arg)	cosine	
tan(arg)	tangent	
cot(arg)	cotangent	
asin(arg)	arc sine	
acos(arg)	arc cosine	
atan(arg)	arc tangent	
acot(arg)	arc cotangent	
sinh(arg)	hyperbolic sine	
cosh(arg)	hyperbolic cosine	
tanh(arg)	hyperbolic tangent	
coth(arg)	hyperbolic cotangent	

Name	Meaning	See
Advanced operators		**Tab. 8.8 on page 91**
classify(int_arg1, real_arg2,real_arg3,arg4)	classification of arg4 into int_arg1 classes	
clip(char_arg1,arg2)	clip arg2 according to char_arg1	
cumul(char_arg1,arg2)	cumulates arg2 according to char_arg1	
flip(char_arg1,arg2)	flip arg2 according to char_arg1	
get_data(char_arg1, char_arg2,char_arg3,arg4)	get data from an external file	
get_experiment(char_arg1, char_arg2,char_arg3,arg4)	include an other experiment	
get_table_fct(char_arg1,arg2)	table function with linear interpolation of table char_arg1 for position arg2	
if(char_arg1,arg2,arg3,arg4)	general purpose conditional if-construct	
mask(char_arg1,arg2,arg3)	mask elements of argument arg21	
matmul(arg1,arg2)	matrix multiplication	
move_avg(char_arg1, char_arg2,int_arg3,arg4)	moving average of running length int_arg3 for arg4	
rank(char_arg1,arg2)	rank of arg2 according to char_arg1	
regrid(char_arg1,arg2)	assign new coordinates to arg2	
run(char_arg1,arg2)	values of arg2 for a single run selected by char_arg1	
run_info(char_arg1)	current run number and/or number of single runs of the current experiment	
transpose(char_arg1,arg2)	transpose arg2 according to char_arg1	
undef( )	undefined element	
Aggregation and moment operators for arguments		**Tab. 8.5 on page 88**
avg(arg)	argument arithmetic mean of values	
avgg(arg)	argument geometric mean of values	
avgh(arg)	argument harmonic mean of values	
avgw(arg1,arg2)	argument weighted mean of values	
count(char_arg1,arg2)	count number of values according to char_arg1	
hgr(char_arg1,int_arg2, real_arg3,real_arg4, arg5)	argument histogram of values	
max(arg)	argument maximum of values	
maxprop(arg)	index of the element where the maximum is reached the first time	
min(arg)	argument minimum of values	
minprop(arg)	index of the element where the minimum is reached the first time	
sum(arg)	argument sum of values	
var(arg)	argument variance of values	
Multiple aggregation and moment operators for arguments		**Tab. 8.6 on page 89**
max_n(arg1 ,..., argn)	maximum per element	
maxprop_n(arg1 ,..., argn)	argument position (1 ... n) where the maximum is reached the first time	
min_n(arg1 ,..., argn)	minimum per element	
minprop_n(arg1 ,..., argn)	argument position (1 ... n) where the minimum is reached the first time	
Dimension related aggregation and moment operators for arguments		**Tab. 8.7 on page 90**
avg_l(char_arg1,arg2)	dimension related argument arithmetic means of values of arg2	
avgg_l(char_arg1,arg2)	dimension related argument geometric means of values of arg2	
avgh_l(char_arg1,arg2)	dimension related argument harmonic means of values of arg2	
avgw_l(char_arg1,arg2,arg3)	dimension related argument weighted means of values of arg2	
count_l(char_arg1,char_arg2, arg3)	dimension related count numbers of values of arg3	

Name	Meaning	See
hgr_l(char_arg1,char_arg2, int_arg3,real_arg4, real_arg5,arg6)	dimension related argument histograms of values of arg6	
max_l(char_arg1,arg2)	dimension related argument maxima of values of arg2	
maxprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the maximum of arg2 is reached the first time	
min_l(char_arg1,arg2)	dimension related argument minima of values of arg2	
minprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the minimum of arg2 is reached the first time	
sum_l(char_arg1,arg2)	dimension related argument sums of values of arg2	
var_l(char_arg1,arg2)	dimension related argument variances of values of arg2	
Multi-run operators (global sensitivity analysis)		**Tab. 8.10 on page 101**
ens(arg)	whole run ensemble	
morris(arg)	get global sensitivity measures for arg	
Multi-run operators (behavioural analysis)		**Tab. 8.11 on page 102**
behav(char_arg1,arg2)	general purpose operator for navigating and aggregating arg2 in the experiment space	
ens(arg)	whole run ensemble	
Multi-run operators (local sensitivity analysis)		**Tab. 8.13 on page 105**
ens(arg)	whole run ensemble	
lin_abs(char_arg1,arg2)	absolute linearity measure	
lin_rel(char_arg1,arg2)	relative linearity measure	
sens_abs(char_arg1,arg2)	absolute sensitivity measure	
sens_rel(char_arg1,arg2)	relative sensitivity measure	
sym_abs(char_arg1,arg2)	absolute symmetry measure	
sym_rel(char_arg1,arg2)	relative symmetry measure	
Multi-run operators (Monte Carlo analysis, global sensitivity analysis and optimization)		**Tab. 8.15 on page 108 Tab. 8.9 on page 100**
avg_e(arg)	run ensemble mean	
avgg_e(arg)	run ensemble geometric mean	
avgh_e(arg)	run ensemble harmonic mean	
avgw_e(arg1,arg2)	run ensemble weighted mean	
cnf(real_arg1,arg2)	positive distance of confidence line from mean avg_e(arg2)	
cor(arg1,arg2)	correlation coefficient between arg1 and arg2	
count_e(char_arg1,arg2)	run ensemble count number of values	
cov(arg1,arg2)	covariance between arg1 and arg2	
ens(arg)	whole run ensemble	
hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	heuristic probability density function	
krt(arg)	kurtosis (4^{th} moment)	
max_e(arg)	run ensemble maximum	
maxprop_e(arg)	run number where the maximum is reached the first time	
med(arg)	median	
min_e(arg)	run ensemble minimum	
minprop_e(arg)	run number where the minimum is reached the first time	
qnt(real_arg1,arg2)	quantile of arg2	
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1	
rng(arg)	range = max_e(arg) - min_e(arg)	
skw(arg)	skewness (3^{rd} moment)	

Name	Meaning	See
stat_full(real_arg1,real_arg2, real_arg3,real_arg4,arg5)	full basic statistical measures	
stat_red(real_arg1,real_arg2, arg3)	reduced basic statistical measures	
sum_e(arg)	run ensemble sum	
var_e(arg)	run ensemble variance	

15.5.2 Experiment Post-Processor Built-In Operators (in Alphabetic Order)

arg	general numerical argument
int_arg	integer constant argument ≥ 0
real_arg	real (float) constant argument
char_arg	character argument

Tab. 15.9 *Experiment post-processor built-in operators (in alphabetical order)*
Monte Carlo operators are also applicable for experiment types uncertainty analysis and Optimization
Monte C. = Monte Carlo analysis, global sensitivity analysis and optimization

Name	Meaning	Type	See	At page
arg1 + arg2	addition	elemental	Tab. 8.3	86
arg1 - arg2	subtraction	elemental	Tab. 8.3	86
arg1 * arg2	multiplication	elemental	Tab. 8.3	86
arg1 / arg2	division	elemental	Tab. 8.3	86
arg1 **arg2	exponentiation	elemental	Tab. 8.3	86
+ arg	identity	elemental	Tab. 8.3	86
- arg	negation	elemental	Tab. 8.3	86
(arg)	parentheses	elemental	Tab. 8.3	86
abs(arg)	absolute value	basic	Tab. 8.4	87
acos(arg)	arc cosine	trigonom.	Tab. 8.4	87
acot(arg)	arc cotangent	trigonom.	Tab. 8.4	87
asin(arg)	arc sine	trigonom.	Tab. 8.4	87
atan(arg)	arc tangent	trigonom.	Tab. 8.4	87
avg(arg)	argument arithmetic mean of values	aggr./mom.	Tab. 8.5	88
avg_e(arg)	run ensemble mean	Monte C.	Tab. 8.9	100
avg_l(char_arg1,arg2)	dimension related argument arithmetic means of values of arg2	aggr./mom.	Tab. 8.7	90
avgg(arg)	argument geometric mean of values	aggr./mom.	Tab. 8.5	88
avgg_e(arg)	run ensemble geometric mean	Monte C.	Tab. 8.9	
avgg_l(char_arg1,arg2)	dimension related argument geometric means of values of arg2	aggr./mom.	Tab. 8.7	90
avgh(arg)	argument harmonic mean of values	aggr./mom.	Tab. 8.5	88
avgh_e(arg)	run ensemble harmonic mean	Monte C.	Tab. 8.9	
avgh_l(char_arg1,arg2)	dimension related argument harmonic means of values of arg2	aggr./mom.	Tab. 8.7	90
avgw(arg1,arg2)	argument weighted mean of values	aggr./mom.	Tab. 8.5	88
avgw_e(arg1,arg2)	run ensemble weighted mean	Monte C.	Tab. 8.9	
avgw_l(char_arg1,arg2, arg3)	dimension related argument weighted means of values of arg3	aggr./mom.	Tab. 8.7	90
behav(char_arg1,arg2)	general purpose operator for navigating and aggregating of arg2 in the experiment space	behav.	Tab. 8.11	102
classify(int_arg1,real_arg2, real_arg3,arg4)	classification of arg4 into int_arg1 classes	advanced	Tab. 8.8	91
clip(char_arg1,arg2)	clip arg2 according to char_arg1	advanced	Tab. 8.8	91
cnf(real_arg1,arg2)	positive distance of confidence line from mean avg_e(arg2)	Monte C.	Tab. 8.15	100
cor(arg1,arg2)	correlation coefficient between arg1 and arg2	Monte C.	Tab. 8.15	108
cos(arg)	cosine	trigonom.	Tab. 8.4	87

Name	Meaning	Type	See	At page
cosh(arg)	hyperbolic cosine	trigonom.	Tab. 8.4	87
cot(arg)	cotangent	trigonom.	Tab. 8.4	87
coth(arg)	hyperbolic cotangent	trigonom.	Tab. 8.4	87
count(char_arg1,arg2)	count number of values	aggr./mom.	Tab. 8.5	88
count_e(char_arg1,arg2)	run ensemble count	Monte C.	Tab. 8.9	100
count_l(char_arg1, char_arg2,arg3)	dimension related count numbers of values of arg3	aggr./mom.	Tab. 8.7	90
cov(arg1,arg2)	covariance between arg1 and arg2	Monte C.	Tab. 8.15	108
cumul(char_arg1,arg2)	cumulates arg2 according to char_arg1	advanced	Tab. 8.8	91
dim(arg1,arg2)	positive difference	basic		87
ens(arg)	whole run ensemble	all types		
exp(arg)	exponential function	basic	Tab. 8.4	87
flip(char_arg1,arg2)	flip arg2 according to char_arg1	advanced	Tab. 8.8	91
get_data(char_arg1, char_arg2,char_arg3,arg4)	get data from an external file	advanced	Tab. 8.8	91
get_experiment(char_arg1, char_arg2,char_arg3,arg4)	include an other experiment	advanced	Tab. 8.8	91
get_table_fct(char_arg1, arg2)	table function with linear interpolation of table char_arg1 for position arg2	advanced	Tab. 8.8	91
hgr(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	argument histogram of values	aggr./mom.	Tab. 8.5	88
hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	heuristic probability density function	Monte C.	Tab. 8.9	100
hgr_l(char_arg1,char_arg2, int_arg3,real_arg4, real_arg5,arg6)	dimension related argument histograms of values of arg6	aggr./mom.	Tab. 8.7	90
if(char_arg1,arg2,arg3,arg4)	general purpose conditional if-construct	advanced	Tab. 8.8	91
int(arg)	integer truncation value	basic	Tab. 8.4	87
krt(arg)	kurtosis (4^{th} moment)	Monte C.	Tab. 8.15	108
lin_abs(char_arg1,arg2)	absolute linearity measure	loc. sens.	Tab. 8.13	105
lin_rel(char_arg1,arg2)	relative linearity measure	loc. sens.	Tab. 8.13	105
log(arg)	natural logarithm	basic	Tab. 8.4	87
log10(arg)	decade logarithm	basic	Tab. 8.4	87
mask(char_arg1,arg2,arg3)	mask elements of argument arg2	advanced	Tab. 8.8	91
matmul(arg1,arg2)	matrix multiplication	advanced	Tab. 8.8	91
max(arg)	argument maximum of values	aggr./mom.	Tab. 8.5	88
max_e(arg)	run ensemble maximum	Monte C.	Tab. 8.9	100
max_l(char_arg1,arg2)	dimension related argument maxima of values of arg2	aggr./mom.	Tab. 8.7	90
max_n(arg1 ,..., argn)	maximum per element	aggr./mom.	Tab. 8.5	89
maxprop(arg)	index of the element where the maximum is reached the first time	aggr./mom.	Tab. 8.5	88
maxprop_e(arg)	run number where the maximum is reached the first time	Monte C.	Tab. 8.15	100
maxprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the maximum is reached the first time of arg2	aggr./mom.	Tab. 8.7	90
maxprop_n(arg1 ,..., argn)	argument position (1 ... n) where the maximum is reached the first time	aggr./mom.	Tab. 8.5	89
med(arg)	median	Monte C.	Tab. 8.15	108
min(arg)	argument minimum of values	aggr./mom.	Tab. 8.5	88
min_e(arg)	run ensemble minimum	Monte C.	Tab. 8.9	

Name	Meaning	Type	See	At page
min_l(char_arg1,arg2)	dimension related argument minima of values of arg2	aggr./mom.	Tab. 8.7	90
min_n(arg1 ,..., argn)	minimum per element	aggr./mom.	Tab. 8.5	89
minprop(arg)	index of the element where the minimum is reached the first time	aggr./mom.	Tab. 8.5	88
minprop_e(arg)	run number where the minimum is reached the first time	Monte C.	Tab. 8.9	100
minprop_l(char_arg1,arg2)	dimension related argument position (1 ... n) where the minimum is reached the first time of arg2	aggr./mom.	Tab. 8.7	90
minprop_n(arg1 ,..., argn)	argument position (1 ... n) where the minimum is reached the first time	aggr./mom.	Tab. 8.5	89
mod(arg1,arg2)	remainder	basic	Tab. 8.4	87
morris(arg)	get global sensitivity measures for arg	glob. sens.	Tab. 8.10	101
move_avg(char_arg1, char_arg2,int_arg3,arg4)	moving average of running length int_arg3 for arg4	advanced	Tab. 8.8	91
nint(arg)	nearest integer value	basic	Tab. 8.4	87
qnt(real_arg1,arg2)	quantile of arg2	Monte C.	Tab. 8.15	108
rank(char_arg1,arg2)	rank of arg2 according to char_arg1	advanced	Tab. 8.8	91
reg(arg1,arg2)	linear regression coefficient to forecast arg2 from arg1	Monte C.	Tab. 8.15	108
regrid(char_arg1,arg2)	assign new coordinates to arg2	advanced	Tab. 8.8	91
rng(arg)	range = max_e(arg) - min_e(arg)	Monte C.	Tab. 8.15	108
run(char_arg1,arg2)	values of arg2 for a single run selected by char_arg1	advanced	Tab. 8.8	91
run_info(char_arg1)	current run number and/or number of single runs of the current experiment	advanced	Tab. 8.8	91
sens_abs(char_arg1,arg2)	absolute sensitivity measure	loc. sens.	Tab. 8.13	105
sens_rel(char_arg1,arg2)	relative sensitivity measure	loc. sens.	Tab. 8.13	105
sign(arg)	sign of value	basic	Tab. 8.4	87
sin(arg)	sine	basic	Tab. 8.4	87
sinh(arg)	hyperbolic sine	trigonom.	Tab. 8.4	87
skw(arg)	skewness (3rd moment)	Monte C.	Tab. 8.15	108
sqrt(arg)	square root	trigonom.	Tab. 8.4	87
stat_full(real_arg1, real_arg2,real_arg3, real_arg4,arg5)	full basic statistical measures	Monte C.	Tab. 8.15	108
stat_red(real_arg1, real_arg2,arg3)	reduced basic statistical measures	Monte C.	Tab. 8.15	108
sum(arg)	argument sum of values	aggr./mom.	Tab. 8.5	88
sum_e(arg)	run ensemble sum	Monte C.	Tab. 8.9	100
sum_l(char_arg1,arg2)	dimension related argument sums of values of arg2	aggr./mom.	Tab. 8.7	90
sym_abs(char_arg1,arg2)	absolute symmetry measure	loc. sens.	Tab. 8.13	105
sym_rel(char_arg1,arg2)	relative symmetry measure	loc. sens.	Tab. 8.13	105
tan(arg)	tangent	trigonom.	Tab. 8.4	87
tanh(arg)	hyperbolic tangent	trigonom.	Tab. 8.4	87
transpose(char_arg1,arg2)	transpose arg2 according to char_arg1	advanced	Tab. 8.8	91
undef( )	undefined element	advanced	Tab. 8.8	91
var(arg)	argument variance of values	aggr./mom.	Tab. 8.5	88
var_e(arg)	run ensemble variance	Monte C.	Tab. 8.9	100
var_l(char_arg1,arg2)	dimension related argument variances of values of arg2	aggr./mom.	Tab. 8.7	90

15.5.3 Character Arguments of Experiment Post-Processor Built-In Operators

Tab. 15.10 summarises for built-in operators character argument values. User-defined operators cannot have pre-defined character argument values.

Tab. 15.10 *Character arguments of experiment post-processor built-in operators*
(*) Character argument can be empty
(**) The length of the character argument from a sequence of digits corresponds to the dimensionality of the non-character and non-constant argument under investigation.

Operator	Argument number	Argument value (without quotation marks, pre-defined values are case-insensitive)	Re-mark
avg_l	1	sequence of digits 0 and 1	(**)
avgg_l	1	sequence of digits 0 and 1	(**)
avgh_l	1	sequence of digits 0 and 1	(**)
avgw_l	1	sequence of digits 0 and 1	(**)
behav	1	(not pre-defined, case insensitive)	(*)
clip	1	(not pre-defined, case insensitive)	
count	1	[all | def | undef]	
count_e	1	[all | def | undef]	
count_l	1	sequence of digits 0 and 1	(**)
count_l	2	[all | def | undef]	
cumul	1	sequence of digits 0 and 1	(**)
flip	1	sequence of digits 0 and 1	(**)
get_data	1	[ascii | netcdf]	
get_data	2	{<directory>/}<file_name>	
get_data	3	{<directory>/}<file_name>	(*)
get_experiment	1	<directory>	
get_experiment	2	<model>	
get_experiment	3	{<directory>/}<file_name>	(*)
get_table_fct	1	{<directory>/}<file_name>	
hgr	1	[bin_no | bin_mid]	
hgr_e	1	[bin_no | bin_mid]	
hgr_l	1	sequence of digits 0 and 1	(**)
hgr_l	2	[bin_no | bin_mid]	
if	1	[< | <= | > | >= | == | != | def | undef]	
lin_abs	1	(not pre-defined, case insensitive)	(*)
lin_rel	1	(not pre-defined, case insensitive)	(*)
mask	1	[< | <= | > | >= | == | !=]	
max_l	1	sequence of digits 0 and 1	(**)
maxprop_l	1	sequence of digits 0 and 1	(**)
min_l	1	sequence of digits 0 and 1	(**)
minprop_l	1	sequence of digits 0 and 1	(**)
move_avg	1	sequence of digits 1 to 9	(**)
move_avg	2	[lin | exp]	
rank	1	[tie_plain | tie_min | tie_avg]	
regrid	1	ascii	
run	1	(not pre-defined, case insensitive)	
run_info	1	[run_nr | nr_of_runs]	
sens_abs	1	(not pre-defined, case insensitive)	(*)
sens_rel	1	(not pre-defined, case insensitive)	(*)
sum_l	1	sequence of digits 0 and 1	(**)
sym_abs	1	(not pre-defined, case insensitive)	(*)
sym_rel	1	(not pre-defined, case insensitive)	(*)
transpose	1	sequence of digits 1 to 9	(**)
var_l	1	sequence of digits 0 and 1	(**)

15.5.4 Constant Arguments of Experiment Post-Processor Built-In Operators

Tab. 15.11 summarises for built-in operators constant argument values.

Tab. 15.11 *Constant arguments of experiment post-processor built-in operators*

Operator	Argument number	Argument type	Argument value restriction
classify	1	int_arg	[0 | ≥ 2]
classify	2	real_arg	[arg2 = arg3 = 0. | arg2 < arg3]
classify	3	real_arg	
cnf	1	real_arg	[0.001 | 0.01 | 0.05 | 0.1]
hgr	2	int_arg	[0 | ≥ 4]
hgr	3	real_arg	[arg3 = arg4 = 0. | arg3 < arg4]
hgr	4	real_arg	
hgr_e	2	int_arg	[0 | ≥ 4]
hgr_e	3	real_arg	[arg3 = arg4 = 0. | arg3 < arg4]
hgr_e	4	real_arg	
hgr_l	3	int_arg	[0 | ≥ 4]
hgr_l	4	real_arg	[arg4 = arg5 = 0. | arg4 < arg5]
hgr_l	5	real_arg	
move_avg	3	int_arg	[0 | ≥ 3]
stat_full	1	real_arg	[0.001 | 0.01 | 0.05 | 0.1] arg1 < arg2
stat_full	2	real_arg	
stat_full	3	real_arg	0. ≤ arg3 < arg 4 ≤ 100.
stat_full	4	real_arg	
stat_red	1	real_arg	[0.001 | 0.01 | 0.05 | 0.1] arg1 < arg2
stat_red	2	real_arg	

15.6 Additionally Used Symbols for the Model and Operator Interface

Tab. 15.12 lists these symbols (subroutine, function and common block names) that are linked in addition to the SimEnv model interface functions in Tab. 5.5 from the object libraries $SE_HOME/lib/libsimenv.a and /usr/local/lib/libnetcdf.a to a Fortran and C/C++ user model when interfacing it to SimEnv. Additionally, the logical unit numbers (luns) 997, 998 and 999 are used.

Tab. 15.12 *Additionally used symbols for the model interface*

Used symbols
csimenv_<string>
isimenv_<string>
jsimenv_<string>
<string>_nc_<string>
nc<string>
nf_<string>
f2c_<string>
c2f_dimids
cdf_routine_name
read_numrecs
write_numrecs

Tab. 15.13 lists these symbols (subroutine, function and common block names) that are linked in addition to the SimEnv operator interface functions in Tab. 8.18 and Tab. 8.19 from the object library $SE_HOME/lib/libsimenv.a to a user-defined experiment post-processing operator.

Tab. 15.13 *Additionally used symbols for the operator interface*

Used symbols
csimenv_<string>
isimenv_<string>
jsimenv_<string>

15.7 Glossary

The glossary defines and/or explains terms in that sense they are used in this User Guide.
An arrow → refers to another term in the glossary.

Adjustment: Numerical modification of a → factor by one of its → sampled values and its → default value during an → experiment. The resulting adjusted value is used instead of the default value of the factor when running the model.

ASCII: The **A**merican **S**tandard **C**ode for **I**nformation and **I**nterchange developed by the American National Standards Institute (http://www.ansi.org) is used in SimEnv to store information in → user-defined files and on request in post-processing output files.

Behavioural analysis: → Experiment type to inspect behaviour of a → model in a space, spanned up by → factors. The factor space is scanned in a deterministic manner, applying deterministically → sampled values of the factors with a flexible scanning strategy for factor sub-spaces.

Coordinate coord: Each → dimension of a → variable and each → operand of an → operator in a → result with a → dimensionality greater than 0 a coordinate is assigned to. A coordinate has a unique name and strictly monotonic ordered coordinate values. The number of coordinate values corresponds to the → extent for this dimension. Consequently, each model output variable with a dimensionality greater than 0 resides at an assigned (multi-dimensional) → grid. Assignments for variables is done in the model output description → user-defined file.

Coupling: → model interface

Cron daemon: The cron daemon runs → shell commands at specified dates and times.

Crontab: The → Unix / → Linux crontab command submits, edits, lists, or removes jobs for the → cron daemon.

Data type: The type of a → variable as declared in the → model and the corresponding model output description → user-defined file. SimEnv data types are byte, short, int, float, and double.

Default value: The nominal (standard) numerical value of an → experiment → factor. The default value is specified in the experiment description → user-defined file and for → the model interface at the language level also in the model code.

Dimension: → dimensionality

Dimensionality dim: The number of dimensions of a model → variable or of an → operator result in → experiment post-processing. In the model output description → user-defined file each variable a dimensionality is assigned to that corresponds to the dimensionality of the related model output field in the model source code. Dimensionality 0 corresponds to a scalar, dimensionality 1 to a vector, dimensionality 2 to a matrix.

Dot script: A sequence of → Unix / → Linux operating system commands stored in an → ASCII file. The sequence of operating system commands is directly interpreted and executed by the → shell. Contrary to → shell scripts a child shell is not spawned. A dot script is preceded by a dot and a space when calling it. All scripts but simenv_put_as and simenv_put_as_simple that can be used in SimEnv within <model>.[ini | run | end] are dot scripts.

Environment variable: At → Unix / → Linux operating system level the so called environment is set up as an array of operating-system and user-defined environment variables that have the form Name=Value. The Value of a Name can be addressed by $Name. In SimEnv use of environment variables in directory strings <direct> is allowed.

Experiment: Performing simulation runs with a → model in a co-ordinated manner by applying → experiment types and running the model in a run ensemble, i.e., a series of single simulation runs.

Experiment post-processing: The work step of processing model output data from the whole run ensemble after performing a simulation → experiment. SimEnv post-processing enables navigation in the →

factor space that is → sampled by an experiment as well as construction of additional output functions by declaration and computation of → results.

Experiment post-processing operator: → operator

Experiment factor: → factor

Experiment type: Pre-defined multi-run simulation experiment. In the process of experiment preparation (defining an experiment by describing it in the experiment description → user-defined file) → factors are assigned to an experiment type and are → sampled in an experiment-specific manner. Currently available experiment types are → global sensitivity analysis, → behavioural analysis, → Monte Carlo analysis, → local sensitivity analysis, and → optimization.

Extent ext: The number of values for a dimension (from the → dimensionality) of a model → variable or of an → operator result in → experiment post-processing. Extents are always greater than 1. Model output variables and operator results of dimensionality 0 do not have an extent.

Expression: → result expression

Factor: Element of the input set of a → model. Factors are manipulated numerically during an → experiment by sampling them. Factors can be addressed in → experiment post-processing and they have there a → dimensionality of 0.

Factor adjustment: → adjustment

Fortran storage model: A rule how to map the elements of a multi-dimensional data field to a 1-dimensional vector and *vice versa*. A multi-dimensional data field field(1:ext_1 , 1:ext_2 ,..., 1:ext_{dim-1} , 1:ext_{dim}) of → dimensionality dim and → extents ext_1, ext_2, ..., ext_{dim-1}, ext_{dim} is mapped in Fortran on a 1-dimensional data field vector(1:ext_1 * ext_2 * ... * ext_{dim-1} * ext_{dim}) in the following way:

```
ipointer = 0
do i_dim = 1 , ext_dim
  do i_dim-1 = 1 , ext_dim-1
    ...
      do i_2 = 1 , ext_2
          do i_1 = 1 , ext_1
              ipointer = ipointer + 1
              vector(ipointer) = field(i_1 , i_2 ,..., i_dim-1 , i_dim)
          enddo
      enddo
    ...
  enddo
enddo
```

For a two-dimensional matrix this storage model corresponds to a column by column storage of the matrix to the vector, starting with the first column and for each column starting with the first row.

GAMS: The **G**eneral **A**lgebraic **M**odeling **S**ystem (http://www.gams.com) is a high-level modeling system for mathematical programming problems. It consists of a language compiler and a number of integrated high-performance solvers. GAMS is tailored for complex, large scale modeling applications, and allows to build large maintainable models that can be adapted quickly to new situations.

Global sensitivity analysis: → Experiment type to determine qualitatively a ranking of the → factors during → experiment post-processing with respect to the factors' sensitivity to a model output. Sensitivity is assessed globally, i.e., for the complete feasibility range of each factor.

Grid: Regular topological structure for a model → variable or an → operator result in → experiment post-processing, spanned up as the Cartesian product of the assigned → coordinates to the variable or the operator result.

IEEE: SimEnv can use on demand for storage of model and post-processor output the **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers (http://www.ieee.org) standard number 754 for binary storage of numbers in floating point representation.

Linux: Linux is a free → Unix-type operating system (http://www.linux.org) originally created by Linus Torvalds with the assistance of developers around the world. SimEnv runs under any Linux implementation for Intel-based hardware and compatibles.

Load Leveler: The load leveler is a network job management system from IBM that handles compute resources. It schedules jobs, and provides functions for building, submitting, and processing them.

Local sensitivity analysis: → Experiment type with an incremental → sample of → factors in the neighbourhood of the → default values of the factors. A local sensitivity analysis in SimEnv is always performed independently for all factors involved. During → experiment post-processing sensitivity, linearity, and symmetry measures can be determined.

Macro: An abbreviation for a unique → result expression to apply during → experiment post-processing. Macros can be embedded into result expressions and are plugged into the expression during its evaluation and computation. Macros are described in the macro description → user-defined file.

Mathematica: Mathematica (http://www.wolfram.com/products/mathematica/introduction.html) seamlessly integrates a numeric and symbolic computational engine, graphics system, programming language, documentation system, and advanced connectivity to other applications.

Matlab: MATLAB (http://www.mathworks.de/products/matlab) is a high-level language for computations and interactive environment for developing algorithms, analysis and visualization of data. It allows to perform computationally intensive tasks faster than with traditional programming languages.

Model: A model is a deterministic or stochastic algorithm, implemented in one or a number of computer programs that transforms a sequence of input values (→ factors) into a sequence of output values (→ variables). Normally, inputs are parameters, driving forces, initial values, or boundary values to the model, outputs are state variables of the model. For many cases, the model will be state deterministic, time and space dependent. For SimEnv, the model, its factors and variables are coupled in the process of → interfacing the model to SimEnv.

Model coupling: → model interface

Model interface: Interfacing a → model to SimEnv means coupling it to SimEnv and enabling finally experimenting with the model within SimEnv. There are coupling interfaces at programming language level for C/C++, Fortran, → Python, Java, → GAMS, → Matlab, and → Mathematica. Additionally, models can be interfaced at the → shell script level by using shell script syntax elements. For all interface techniques the interfaced model itself has to be wrapped into a shell script.

Model output variable: → variable

Monte Carlo analysis: → Experiment type with pre-single run perturbations of experiment → factors. For each perturbed factor a → probability density function pdf with function parameters is assigned to. During the → experiment → adjustments of the factors are realizations from the pdf's using random number techniques. In → experiment post-processing statistical measures can be derived from model output of the run ensemble. A prominent statistical measure is the heuristic pdf (histogram) of a model → variable and its relation to the pdf's of the factors.

NetCDF: **Net**work **C**ommon **D**ata **F**orm is an interface for array-oriented data access and a library that provides an implementation of the interface. The NetCDF library also defines a machine-independent format for representing scientific data. Together, the interface, library, and format support the creation, access, and sharing of scientific data. The NetCDF software was developed at the Unidata Program Center in Boulder, Colorado. NetCDF is freely available. SimEnv follows for model and → experiment post-processing output storage the NetCDF Climate and Forecast (CF) metadata convention 1.0 (http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html) and extends it. To dump NetCDF files, ncdump is used by SimEnv.

OpenDX: The **Open D**ata E**x**plorer OpenDX (http://www.opendx.org) is a uniquely full-featured open source project and software package for the visualization of scientific, engineering and analytical data: Its open system design is built on a standard interface environment. The data model provides users with great flexibility in creating visualizations. OpenDX is based on IBM's Visualization Data Explorer.

Operand: Argument of an → operator in SimEnv → experiment post-processing. An operand can be a model → variable, an experiment → factor, a constant, a character string, → a macro and an operator.

Operator: Computational algorithm how to transform the values of a sequence of → operands into the values of the operator result during → experiment post-processing. An operator transforms → dimensionality, → extents, and → coordinates from the operands into the corresponding information for the operator result. There are built-in elemental, basic, and advanced operators as well as built-in operators related to specific → experiment types. Additionally, SimEnv offers specification of user-defined operators according to an operator interface. User-defined operators are announced to the system in the operator description → user-defined file.

Optimization: → Experiment type to minimize a cost function (objective function) over a bounded → factor space. In SimEnv a simulated annealing strategy (cf. Section 4.6 for explanation) is used to optimize the cost function that is formed from model → variables. Often the cost function represents a distance between model output and reference data to find an optimal point in the factor space that fits best the model behaviour with respect to the reference data.

Parallel Operating Environment: → POE

POE: The **P**arallel **O**perating **E**nvironment POE from IBM supplies services to allocate nodes, assign jobs to nodes and launch jobs on a compute cluster.

Probability density function pdf: A probability density function serves to represent a probability distribution in terms of integrals. A probability distribution assigns to every interval of real numbers a probability.

Python: Python (http://www.python.org) is a portable, interpreted, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes.

Result: In SimEnv → experiment post-processing a result (synonym: output function) is derived from model output of the → experiment and from reference data. A result is specified by a result expression, optionally prefixed by a result description and a result unit string.

Result expression: A chain of → operators from built-in or user-defined operators applied to model output → variables and/or reference data. A result expression is a part of an → experiment post-processing → result.

Sample: A set of numerical → factor values created during experiment preparation.

Shell: A shell is the command interpreter for the operating systems → Unix and → Linux.

Shell script: A sequence of → Unix / → Linux operating system commands stored in an → ASCII file. A shell script is interpreted and executed by a → shell. Contrary to → dot scripts a child shell is spawned when calling a shell script that inherits the → environment variables of the father (calling) shell. After returning to the father shell it does not transfer the environment variables and other variables of the child shell to the father shell. SimEnv demands the Bourne shell sh.

SimEnvVis: The visualization framework of SimEnv. It does not belong to the standard distribution of SimEnv. Contact the SimEnv developers to get SimEnvVis.

Simulation: Performing → experiments with → models

Unix: A computer operating system (http://www.unix.org), originally developed at AT&T/USL. SimEnv runs under the AIX Unix implementation for RS6000 hardware and compatibles from IBM.

User-defined files: A set of → ASCII files to describe → model, → experiment, → operator, → macro, and → GAMS model specific information and to determine general SimEnv settings. All user-defined files follow the same syntax rules.

Variable: Element of the output set of a → model that is stored during a SimEnv experiment in SimEnv model output. Variables are defined in the model output description → user file. Each variable has a unique → data type, a → dimensionality, → extents and an assigned → grid. Normally, a variable consists of a series of values, forming a field.

White spaces: → (also known as blanks) ASCII characters space and horizontal tabulator used in → user-defined files or within result expressions in → experiment post-processing.

Workspace: The directory, a SimEnv service was started from.