The Multi-Run Simulation Environment

SimEnv

User Guide for Version 3.1 (Jan 25, 2013)

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 3.1 (Jan 25, 2013)

by

Michael Flechsig	flechsig@pik-potsdam.de
Uwe Böhm	uwe.boehm@dwd.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

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 brought out into the open.

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

Contents

Tables

Figures

Examples

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 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 experiment output and reference data to navigate and post-process in the combined sample and experiment 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 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 (nominal) 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}**. Experiment output as the set of all 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 – elementary effect method
  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.
- Global sensitivity analysis – variance based method
  Identify contribution of each factor to the variance of the model output.
  For determination of most important factors and those factors that can be fixed at any value over the range of uncertainty without significantly modifying the output uncertainty.
- Deterministic factorial design
  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.
- Uncertainty analysis – 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.
- 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.

- Bayesian technique – Bayesian calibration
  Reduce uncertainty about factor values by deriving a representative sample from factor prior distributions while having measurement values from the system available for model – measurement comparison.
  For uncertainty analysis and reduction with a Bayesian technique.
- Optimization – Simulated Annealing
  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 experiment output post-processing enables navigation in the complex factor – experiment output space and interactive filtering of experiment 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.

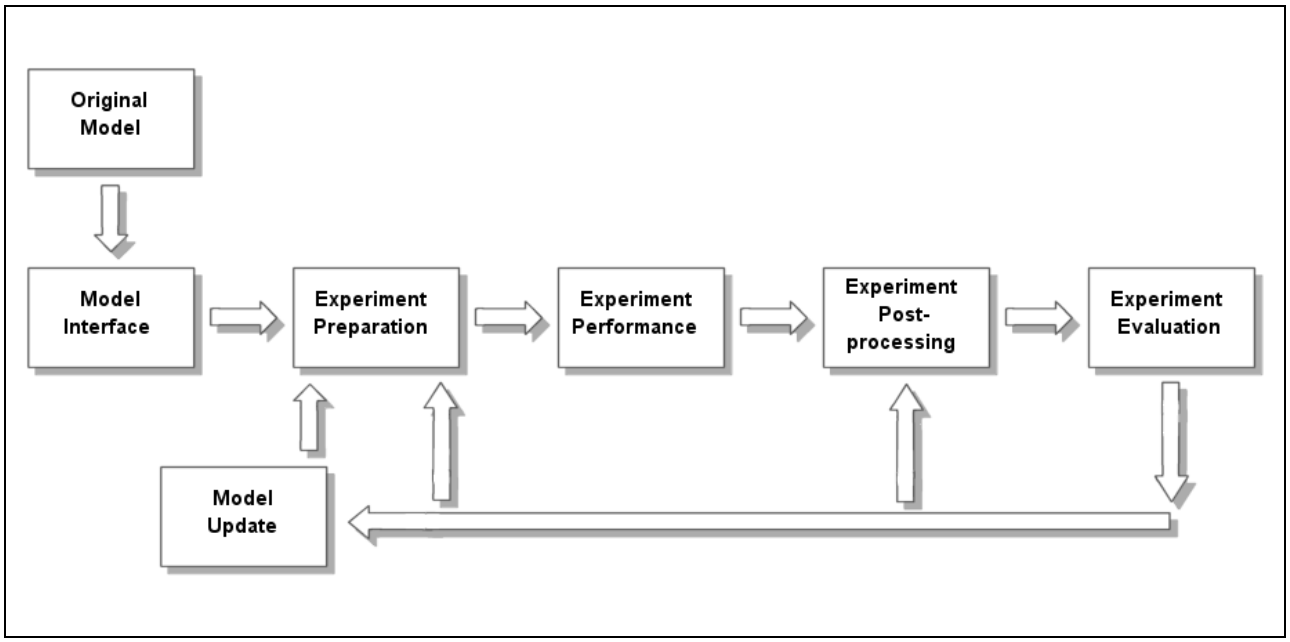


Fig. 0.1 *SimEnv system design*

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.
- Visual experiment evaluation:
  For post-processed experiment 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
italic	a keyword or a sub-keyword in user-defined files
" ... "	double quotation marks enclose a value string 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
<attr_name>	NetCDF attribute name
<co_name>	coordinate name as defined in <model>.mdf or generated automatically (see Tab. 11.11)
<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 (nominal) 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>	sampled value of a factor from <model>.smp
<file_name>	name of a ASCII data file
<model>	model name to start a SimEnv service with
<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_byte>	integer*1 / byte value (e.g., -123)
<val_short>	integer*2 / short value (e.g., -12345)

Placeholder	Description
<val_int>	integer*4 / int value (e.g., -123456)
<val_float>	real*4 / float value in integer*4 / int (e.g., -123456), fixed point (e.g., -123.456) or floating point (scientific) (e.g., -1.23456e+2) notation
<val_double>	real*8 / double value in integer*4 / int (e.g., -123456789), fixed point (e.g., -123.456789) or floating point (scientific) (e.g., -1.23456789d+2) notation
<val_list>	list of values in explicit or implicit notation according to Tab. 12.5
<var_name>	variable name as defined in <model>.mdf or a built-in variable (see Tab. 11.9)
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)
dim	dimensionality of a variable, operand or result
ext(i) (i=1,…,dim)	extent of dimension i of a variable, operand or result
coord(i) (i=1,…,dim)	coordinate of dimension i of a variable, operand or result
no._of_...	number of …

1.2 Example Layout

All examples in this document but for GAMS refer to a hypothetical global atmosphere – biosphere simulation **model world**. It describes dynamics of atmosphere and biosphere at the global scale over 200 years. Global lateral (latitudinal and longitudinal) model resolution is 4° x 4° between 178°W and 178°E and/or 88°N and 88°S (exceptions see below), 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 56°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. 11.4).

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 217.
- 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 11, for the complete environment check Tab. 11.15 on page 201. 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 7:

<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 51, for a GAMS model example check Section 5.7 on page 52.
- 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_DFD_c 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_DFD_c**	post-processor result sequence

  Either
  - Start

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

    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)

- Experiment output post-processing will be started for this experiment
  - With the post-processing input file world_post_DFD_c 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 to the PIK visualization server check in Section 11.2 on page 186 the SimEnv service

simenv.key	<user_name>

3 Version 3.1

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 3.1*

Type	Check / see	On page	Description
update	Section 5.4 Section 8.5.1	45 154	Link scripts for models and user-defined operators $SE_HOME/lib/simenv_[mod | opr]_[f | c | cpp].lnk renamed to $SE_HOME/lib/link_simenv_[mod | opr]_[f | c | cpp].sh
			Bug fixes

Tab. 3.2 *User actions to upgrade to Version 3.1*

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

Tab. 3.3 *SimEnv installations*

Organization	Machine	Cluster login node	SE_HOME=
Potsdam Institute for Climate Impact Research (PIK) www.pik-potsdam.de	cluster.pik-potsdam.de	login01, login02	/iplex/01/sys/applications/simenv
PIK www.pik-potsdam.de	viss02.pik-potsdam.de	--	/iplex/01/sys/applications/simenv
PIK www.pik-potsdam.de	bs08.pik-potsdam.de	--	/usr/local/simenv
TU Berlin www.evur.tu-berlin.de	drawe.evur.tu-berlin.de	drawe	/opt/simenv
Global Climate Forum www.globalclimateforum.org		--	/opt/simenv

3.2 Limitations / Problems and Their Workarounds

Tab. 3.4 *Limitations / problems and their workarounds in Version 3.1*

Where Limitation / Problem Workaround	Description
Where	Overall
Limitation	Current SimEnv technical limitations as specified in Tab. 15.3 on page 219
Workaround	None
Where	Overall but visual result evaluation with SimEnvVis
Limitation	Without graphical user interface
Workaround	None
Where	Experiment performance: Experiment output to NetCDF
Problem	Check on undefined experiment output results in noticeably 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 experiment output and additionally 200 sec for common nc file output.
Workaround	Specify in <model>.cfg for the sub-keyword *message_level* the value = “error” (see also Section 11.3)
Where	Experiment performance: Under Distributed Resource Manager DRM control in distributed sub-mode dis and on PIK machine bs08:
Limitation	Model output is not checked on undefined values on machine bs08 in general and on the Iplex compute cluster under load leveler control in in sub-mode dis, even when specified by the sub-keyword *message_level* in <model>.cfg.
Workaround	Perform experiment at login node or under load leveler DRM control in an other sub-mode

3.3 Known Bugs and Their Workarounds

Tab. 3.5 *Known bugs and their workarounds in Version 3.1*

Where Bug Workaround	Description
Where	Experiment performance: Experiment type UNC_MC with stopping rule under Distributed Resource Manager DRM control in distributed sub-mode dis or on the login machine distributed on a multicore processor machine
Bug	Experiment may not come to an end
Workaround	Use an other experiment performance setting
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 post-processing: Operators, clip, mask, mask_file, unmask_file
Bug	Do not work with the coordinate range c = <coordinate_value$_1$> { : <coordinate_value$_2$> }
Workaround	Use index range i= <index_value$_1$> { : <index_value$_2$> } instead

4 Experiment Types

SimEnv supplies a set of pre-defined multi-run experiment types. Each experiment type addresses a special experiment method 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

SimEnv supplies a set of pre-defined multi-run experiment types, where each type addresses a special multi-run experiment method 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

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

with

ST	state transition description
Z	state variables' vector
P	parameter vector
IX	input (driving forces) vector
Z_0	initial value vector
t	time
Δt	time increment
m	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* and initial values Z_0. 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 parametrising the shape of a driving force

When this has been done, the model behaviour finally depends only on the parameters *P* and the initial values Z_0. From the methodical point of view there is no difference between parameters and initial values because all are considered as constant during one model run. That is why in SimEnv the three model components parameters, drivers and initial 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:

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

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

$$X_{k,n} = \begin{pmatrix} x_{11} & \dots & x_{1k} \\ \dots & & \dots \\ x_{n1} & \dots & x_{nk} \end{pmatrix} = (\hat{}X_1 ,..., \hat{}X_n)^T \qquad k > 0,\ n > 0$$

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:

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

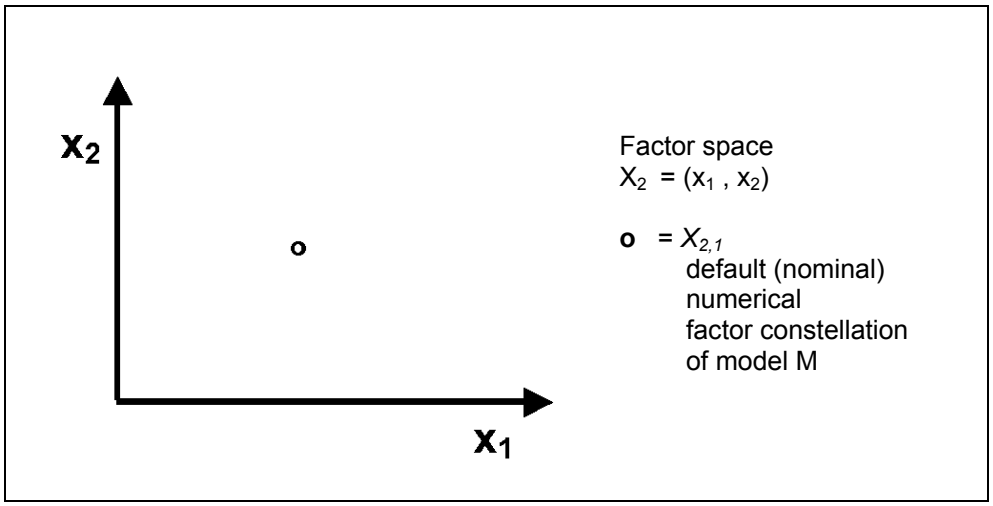


Fig. 4.1 *Factor space*

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 probabilistic sampling strategies that offer a broad range of techniques for

- Experimentation with models
- Post-processing experiment output
- Interpreting results with respect to uncertainty and sensitivity matters of models.

The experiment types are described in detail in the following Sections. Tab. 4.1 provides an overview on the experiment types. Is is ordered in a sequence which may for many cases is well suited for assessing any model.

Tab. 4.1 *SimEnv experiment types*

Experiment type	Description
GSA_EE: Global sensitivity analysis – elementary effects method	Qualitative ranking of a large number of factors with respect to their sensitivity on model output derived from statistical measures of local elementary effects at randomly selected trajectories in the factor space. For determination of the most important factors.
GSA_VB: Global sensitivity analysis – variance based	Identify contribution of each factor to the variance of the model output. For determination of most important factors and those factors that can be fixed at any value over the range of uncertainty without significantly modifying the output uncertainty.
DFD: Deterministic factorial design	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.
UNC_MC: Uncertainty analysis – Monte Carlo method	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.
LSA: 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.
BAY_BC: Bayesian technique – Bayesian calibration	Reduce uncertainty about factor values by deriving a representative sample from factor prior distributions while having measurement values from the system available for model – measurement comparison. For uncertainty analysis and reduction with a Bayesian technique.
OPT_SA: Optimization – simulated annealing	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.

The experiment types come with special properties.

- A model-free method is model independent. It can cope with non-linear, non-additive and non-monotonic models. Each of these three properties holds for the relation between each factor x_i (i=1 ,.., k) and the state variable and/or output function under consideration.
- Some methods take the sample from pre-defined factor levels rather than from pre-defined factor distributions. The former approach allows for performing an analysis without knowledge about factor distributions and the accompanying distribution parameters.
- A method supports the factor range of each factor globally if the method takes into account the whole range of the factor variation and for probabilistic methods (where samples are taken from distributions) the whole probability density function is explored.
- A method supports multi-factor variation if it investigates into all factors rather than focusing on for instance turn on single factors. The latter approach disregards interaction effects between factors on the model output and finally on the derived measures.
- Often, modelers and or analysts are confronted with correlated factors. Even if specification of an experiment in SimEnv does not explicitly support correlated samples (for example, method UNC_MC) it allows to import such samples.
- The computational costs of a an experiment type is the number of single model runs necessary to perform the method. Computational costs may limit applicability of an individual experiment type if the computational costs in terms of CPU or wall clock time consumption of a single simulation run are high.

Tab. 4.2 *Experiment type characteristics*
N_{Mc} denotes the number of Monte Carlo runs

Experiment type	Model free?	Sample taken from	Factor range + multi-factor variation	Cope with correlated factors?	Computational costs (for k factors)
GSA_EE	yes	levels	global + multi-dimensional	no	10 * (k+1) + 1
GSA_VB	yes	distributions	global + multi-dimensional	no	N_{Mc} * (k+2) + 1
DFD	yes	levels	(global + multi-dimensional)	yes	design dependent
UNC_MC	yes	distributions	global + multi-dimensional	yes	N_{Mc}+1
LSA	yes	levels	local	no	2*k+1
BAY_BC	yes	distributions	global + multi-dimensional	yes	N_{Mc}+1 (N >> 0)
OPT_SA	yes	distributions	global + multi-dimensional	(yes)	unpredictable

4.2 Global Sensitivity Analysis – Elementary Effects Method GSA_EE

The guiding philosophy of a GSA_EE experiment is to determine those factors that influence a model state z the most and to distinguish them from factors that are negligible. Contrary to a local sensitivity analysis LSA during this experiment type the entire space where the factors may vary is considered.
The GSA_EE experiment in SimEnv applies the method of Morris (1991) in its modifications by Campolongo *et al.* (2005), Saltelli *et al.* (2008) and Sin & Gernaey (2009). Its main approach is to derive qualitative global sensitivity measures for all factors by computing a statistics on a set of local sensitivity measures, the so-called elementary effects. The result of this analysis is a ranking of the factors in terms of their importance with respect to the model state z.

The modified Morris method is as follows (cf. also Fig. 4.3):

- Specify for each factor x_i (i=1 ,…, k) a distribution with its distribution parameters.
- Span up in the factor space X_k a k-dimensional p-level grid by determining for each factor x_i p equidistant quantiles q_1 ,…, q_p. These quantiles vary in [0,1] and the corresponding individual factor values in the factor space follow their own distributions. In the corresponding k-dimensional factor quantile space QX_k this results in a regular equidistant grid.
- Specify in the factor quantile space QX_k a unique jump width Δ as a value from { 1/(p-1) , 2/(p-1) , …, (p-2)/(p-1) }. The jump width specifies how far away in terms of equidistant quantiles another point is selected from an existing grid point to compute an elementary effect as described below. In the factor space X_k the unique jump width Δ transforms for each factor x_i to values Δ_{im} where m corresponds to the value of x_i of the existing grid point.
- Select at the p-level grid in the factor space randomly a starting grid point $x^0 = (x_1 ,\ldots, x_k)$ and randomly another grid point $x^1 = (x_1 ,\ldots, x_{i-1} , x_i+\Delta_{im} , x_{i+1} ,\ldots, x_k)$ that differs from x^0 in exactly one factor x_i by Δ_{im} (i=1 ,…, k).
- Compute from these two grid points the elementary effect of the factor x_i for the model state variable z by

  $$d_i(x,z) = z(x_1 ,\ldots, x_{i-1} , x_i+\Delta_{im} , x_{i+1} ,\ldots, x_k) – z(x_1 ,\ldots, x_k) / \Delta_{im}$$

- Proceed by randomly selecting a new grid point x^2 to the previous grid point x^1 for another elementary effect $d_j(x,z)$ (i≠j) until k+1 points { x^0 , x^1 , …, x^k } are sampled. Such a series of k+1 points is called a trajectory. For one trajectory k elementary effects $d_i(x,z)$ (i=1 ,…, k) (one for each factor) can be determined by two consecutive points x^i and x^{i+1} (i=0 ,…, k).
- Determine randomly r trajectories in this way finally resulting in r elementary effects $d_i(x,z)$ for each factor x_i.

  With

  σ_{xi}^2 = run ensemble variance of factor values x_i (i=1 ,…, k)

  and

  σ_z^2 = run ensemble variance of variable z

  consider the distributions (i=1 ,…, k)

  F_i^{abs} = { $|d_i(x,z)\cdot\sigma_{xi}/\sigma_z|$ } and compute mean $\mu_i^{abs} = \Sigma\, |d_i(x,z)| / r$

  F_i = { $d_i(x,z)\cdot\sigma_{xi}/\sigma_z$ } and compute variance $\sigma_i^2 = \Sigma\, (d_i(x,z) – \Sigma d_i(x,z) / r)^2 / (r–1)$

- 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 σi with respect to the other σj
    indicates that the factor xi is involved in interactions with other factors or
    indicates that the effect of xi on the model state z is non-linear

  (Saltelli *et al.*, 2004). See also Fig. 4.2.

Note the following remarks:

- According to Saltelli *et al*. (2004) and as a rule of thumb
  - The number of levels p of the grid should by even and should be 4 or 6
  - The optimal unique jump width Δ in the factor quantile space is (p/2)/(p-1)
  - The number of trajectories r should range around 10
- According to Sin and Gernaey (2009)
  - The sensitivity measures μ_i^{abs} and σ_i^2 are non-dimensional as they are scaled by σ_{xi} and σ_z. This allows for applying the Morris method with factors that significantly differ in their value ranges, i.e., for factors with large values together with factors with small values. Additionally, the scaling allows for comparing the sensitivity measures between different state variables z and/or output functions.

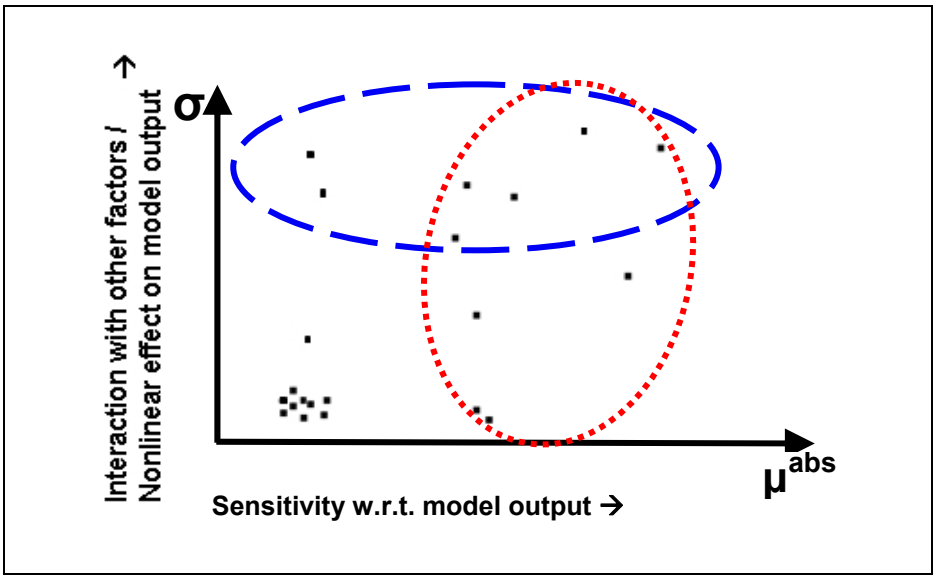
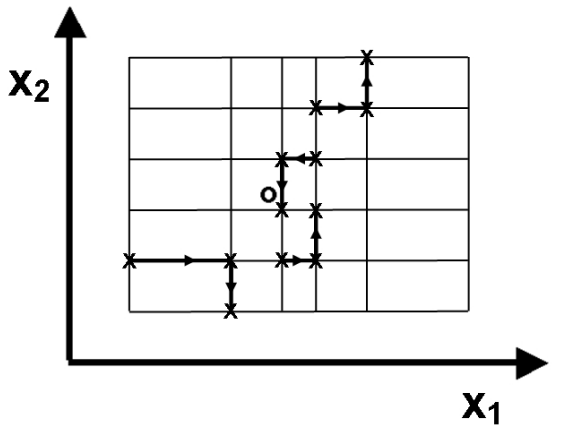
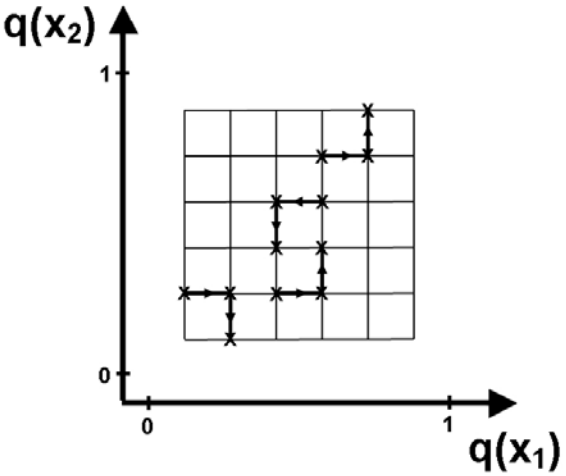
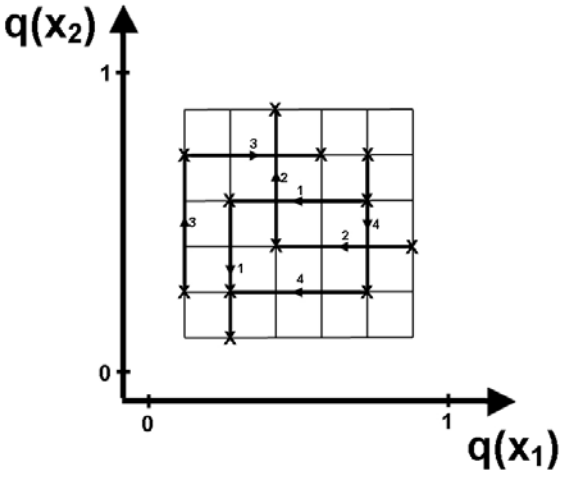


Fig. 4.2 *Result plot for GSA_EE of 21 factors. Factors bounded by the dotted line were identified as sensitive, factors bounded by the dashed line show interactions with other factors and/or a nonlinear effect on model output.*







{**x**} = $X_{2,12}$
sample of size 12
representing
4 trajectories
at a 6-level grid
assuming a
normal distribution for x_1
and a
uniform distribution for x_2

o = $X_{2,1}$
default (nominal)
numerical
factor constellation
of model M

upper figure:
sample for Δ = 1/5
in the 2-dimensional
factor space
$X_2 = (x_1 , x_2)$

middle figure:
sample for Δ = 1/5
in the factor quantile
space
$QX_2 = (q(x_1) , q(x_2))$

lower figure:
sample for optimal
Δ = 3/5
in the factor quantile
space
$QX_2 = (q(x_1) , q(x_2))$
(numbers indicate
individual trajectories)

The arrows indicate the sequence how each trajectory was generated from sampling points.

Fig. 4.3 *Sample for GSA_EE*

4.3 Global Sensitivity Analysis – Variance-Based Method GSA_VB

The guiding question of variance based methods for sensitivity analysis of model output is how the variance of model output depends on the variability of the model factors and how the output variance can be apportioned accordingly. With respect to this approach, variance based methods have a connection to analysis of variance (ANOVA) studies and the design of experiments (DOE). For variance based methods it is assumed that all what is known about the sensitivity of the model output to the factors – and consequently can be exploited – is captured by the variance of the model output.

For variance decomposition the method of Sobol' (Sobol, 1993) and a cost-efficient implementation by Saltelli (Saltelli, 2002, Saltelli *et al.*, 2008) have been implemented.

Related to the decomposition of the variance the overall goal is to decompose the variance V(z) on model output z into a sum of 2^k -1 terms

$$V(z) = \sum V_i + \sum\sum V_{ij} + \ldots + \sum\ldots\sum V_{m\ldots s} + \ldots + V_{12\ldots k} \qquad (1 \le i < j < \ldots < m \ldots < s \ \ldots \le k$$

k = number of factors)

with

V_i = fraction caused by factor x_i
V_{ij} = fraction caused by interaction of factors x_i and x_j
…
$V_{m\ldots s}$ = fraction caused by interaction of factors x_m ,…, x_s
…
$V_{12\ldots k}$ = fraction caused by interaction of all k factors

Theoretically, the (unconditional) variance V(Z) of Z can be reduced by fixing X_i to its real value x_i*:

$V_{-Xi}(Z|X_i=x_i^*)$ conditional residual variance of Z when fixing X_i to any x_i*

where uppercase X_i and Z indicate random variables, lowercase x_i and z values of a random variable and the subscript $-X_i$ stands for "all but X_i"

As the real value x_i* of factor X_i is unknown consider instead the average residual variance:

$E_{Xi}(V_{-Xi}(Z|X_i=x_i^*))$ average residual variance of Z that can be obtained when fixing X_i

The smaller this value the more influential is factor X_i. It always holds:

$$E_{Xi}(V_{-Xi}(Z|X_i=x_i^*)) \le V(Z)$$

Taking into account the variance decomposition rule for conditional distributions:

$$V(Z) = V_{Xi}(E_{-Xi}(Z|X_i=x_i^*)) + E_{Xi}(V_{-Xi}(Z|X_i=x_i^*))$$
$$\text{(abbr.)} = V(E(Z|X_i)) + E(V(Z|X_i))$$
$$= \text{main effect of } X_i \text{ on } Z + \text{residual}$$

Dividing the above equation by V(Z) results in:

$S_i = 1 - E(V(Z|X_i)) / V(Z)$ **1st order sensitivity index**
$= V(E(Z|X_i)) / V(Z)$
$\epsilon [0,1]$

S_i indicates by how much V(Z) would be reduced on average if factor X_i could be fixed. The larger S_i the more influential is factor X_i.

In parallel, $z(x_1, \ldots, x_k)$ can be developed in a high dimensional model representation HDMR (Rabitz et al., 1999) of 2^k terms as

$$z(x_1, \ldots, x_k) = z_0 + \Sigma z_i + \Sigma\Sigma z_{ij} + \ldots + \Sigma\ldots\Sigma z_{m\ldots s} + \ldots + z_{12\ldots k} \quad (1 \le i < j < \ldots < m \ldots < s \ \ldots \le k)$$

with

z_0 = constant
z_i = $z_i(x_i)$
z_{ij} = $z_{ij}(x_i,x_j)$
…
$z_{m\ldots s}$ = $z_{m\ldots s}(x_m, \ldots, x_s)$
…
$z_{1\ldots k}$ = $z_{1\ldots k}(x_1, \ldots, x_k)$

This decomposition is unique provided that the factors $x_1, \ldots, x_k$ are orthogonal. A necessary condition for orthogonality of the factors $x_1, \ldots, x_k$ is that they are uncorrelated. With this it holds the following ANOVA-HDMR

$$V(Z) = \Sigma V_i + \Sigma\Sigma V_{ij} + \ldots + \Sigma\ldots\Sigma V_{m\ldots s} + \ldots + V_{12\ldots k}$$

and it can be shown that

V_i = $V(E(Z|X_i))$
V_{ij} = $V(E(Z|X_iX_j)$ - $(V_i + V_j)$
…
$V_{m..s}$ = $V(E(Z|X_{m\ldots s})$ - Σ $V_{m\ldots s \text{ of lower order}}$
…
$V_{1\ldots k}$ = $V(E(Z|X_{1\ldots k})$ - Σ $V_{1\ldots k \text{ of lower order}}$

Dividing the ANOVA-HDMR by V(Z):

$$1 = \Sigma S_i + \Sigma\Sigma S_{ij} + \ldots + \Sigma\ldots\Sigma S_{m\ldots s} + \ldots + S_{12\ldots k} \qquad \textbf{(*)}$$

it holds

S_i = $V(E(Z|X_i)) / V(Z)$ — **1st order sensitivity index**
S_{ij} = $V(E(Z|X_iX_j) / V(Z)$ - $(S_i + S_j)$ — **2nd order sensitivity index**
…
$S_{m..s}$ = $V(E(Z|X_{m\ldots s}) / V(Z)$ - $\Sigma S_{m\ldots s \text{ of lower order}}$
…
$S_{1\ldots k}$ = $V(E(Z|X_{1\ldots k}) / V(Z)$ - $\Sigma S_{1\ldots k \text{ of lower order}}$

$S_{m\ldots s}$ indicates by how much V(Z) would be reduced on average if $x_m, \ldots, x_s$ could be fixed.
The number of model evaluations (model runs) necessary to compute the complete set of sensitivity indices is

$C = N_{Mc}*N_{Mc}*(2^k-1)$

where N_{Mc} is the Monte Carlo sample size.

Saltelli defined another sensitivity index, the total sensitivity index as:

S_{Ti} = sum of all terms in **(*)** that include factor X_i — **total effect index**

and showed that it can be computed by

$S_{Ti} = 1 - V(E(Z|X_{-i})) / V(Z)$
$= E(V(Z|X_{-i})) / V(Z)$

S_{Ti} is a measure for the total contribution to the output variation due to factor x_i. The smaller S_{Ti} ($S_{Ti} \approx 0$) the more the factor x_i can be fixed at any value over its range of uncertainty without significantly modifying the output uncertainty expressed by V(Z).

The following properties apply for 1st order sensitivity indices S_i and total effect indices S_{Ti}:

$0 \le S_i \le 1$		numerical irregularities may occur (orthogonality!)
ΣS_i	≤ 1	ditto
ΣS_i	$= 1$	for additive models (without interaction effects) and orthogonal (uncorrelated) factors.
S_{Ti}	≥ 0	numerical irregularities may occur (orthogonality!)
ΣS_{Ti}	≥ 1	ditto
ΣS_{Ti}	$= 1$	for additive models (without interaction effects) and orthogonal (uncorrelated) factors.
S_{Ti}	$\ge S_i$	
S_{Ti}	$= S_i$	if x_i is **not** involved in interactions with other factors

To determine the 1st order sensitivity indices S_i and total effect indices S_{Ti} Saltelli (Saltelli, 2002, Saltelli *et al.*, 2008) proposed the following cost efficient method:
For k factors x_1 ,…, x_k draw two samples of size n according to the marginal distributions of the factors: a sample S and a resample R. Construct for each factor x_i (i = 1 ,…, k) a resulting sample RS_i by copying all values from the resample R but for factor x_i. For factor x_i take the values from the sample S. In other words: do not re-sample for RS_i the factor x_i that first order and/or total effect index is to be estimated.

$$\begin{pmatrix} s_{11} & \cdots & s_{1k} \\ \vdots & & \vdots \\ s_{n1} & \cdots & s_{nk} \end{pmatrix} \quad \begin{pmatrix} r_{11} & \cdots & r_{1k} \\ \vdots & & \vdots \\ r_{n1} & \cdots & r_{nk} \end{pmatrix} \quad \begin{pmatrix} r_{11} & \cdots & r_{1i-1} & s_{1i} & r_{1i+1} & \cdots & r_{1k} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ r_{n1} & \cdots & r_{ni-1} & s_{ni} & r_{ni+1} & \cdots & r_{nk} \end{pmatrix} = \begin{pmatrix} rs_{11}^{(i)} & \cdots & rs_{1k}^{(i)} \\ \vdots & & \vdots \\ rs_{n1}^{(i)} & \cdots & rs_{nk}^{(i)} \end{pmatrix}$$

Sample S Resample R Resulting sample RS_i

Run model for the sample S, the re-sample R, and the k resulting samples RS_i to get

$z(S) = (z_1^{(S)}, \ldots, z_n^{(S)}) = (z(s_{11},\ldots, s_{1k}),\ldots, z(s_{n1},\ldots, z(s_{nk}))$
$z(R) = (z_1^{(R)}, \ldots, z_n^{(R)}) = (z(r_{11},\ldots, r_{1k}),\ldots, z(r_{n1},\ldots, z(r_{nk}))$
$z(SR_i) = (z_1^{(SRi)}, \ldots, z_n^{(SRi)}) = (z(sr_{11}^{(i)},\ldots, sr_{1k}^{(i)}),\ldots, z(sr_{n1}^{(i)},\ldots, z(sr_{nk}^{(i)}))$

To estimate S_i: use model output from S and RS_i $$S_i = \frac{z(S) \circ z(RS_i)/n - \bar{z}^2(S)}{y(S) \circ y(S)/n - \bar{z}^2(S)}$$

To estimate S_{Ti}: use model output from R and RS_i $$S_{Ti} = 1 - \frac{z(R) \circ z(RS_i)/n - \bar{z}^2(R)}{z(R) \circ z(R)/n - \bar{z}^2(R)}$$

with ∘ = scalar product

$$z(A) \circ z(B) = \sum_{i=1}^{n} z_i^{(A)} \cdot z_i^{(B)}$$

$$\bar{z}(A) = \frac{1}{n} \sum_{i=1}^{n} z_i^{(A)}$$

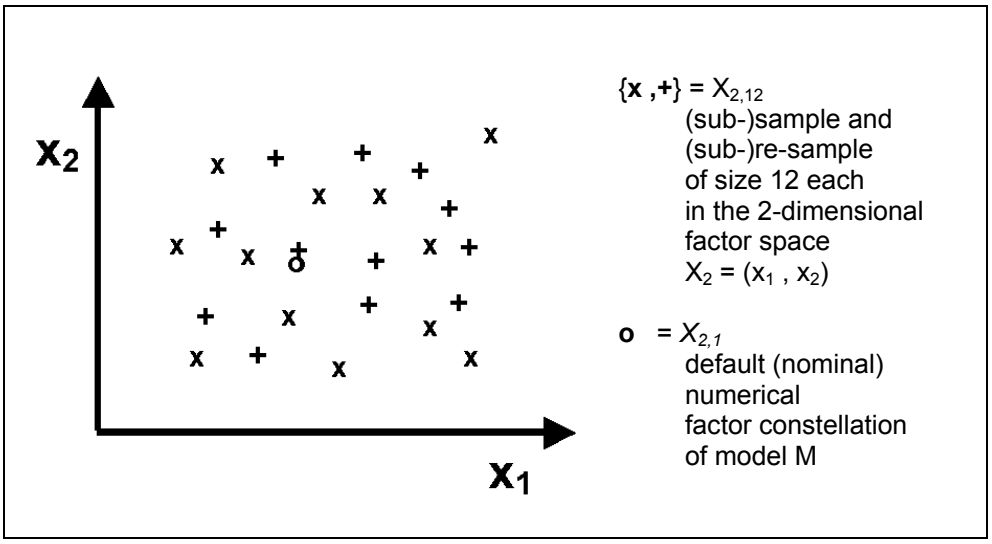


Fig. 4.4 *Sample for GSA_VB*

4.4 Deterministic Factorial Design DFD

Deterministic Factorial Design DFD 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.

A DFD experiment 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

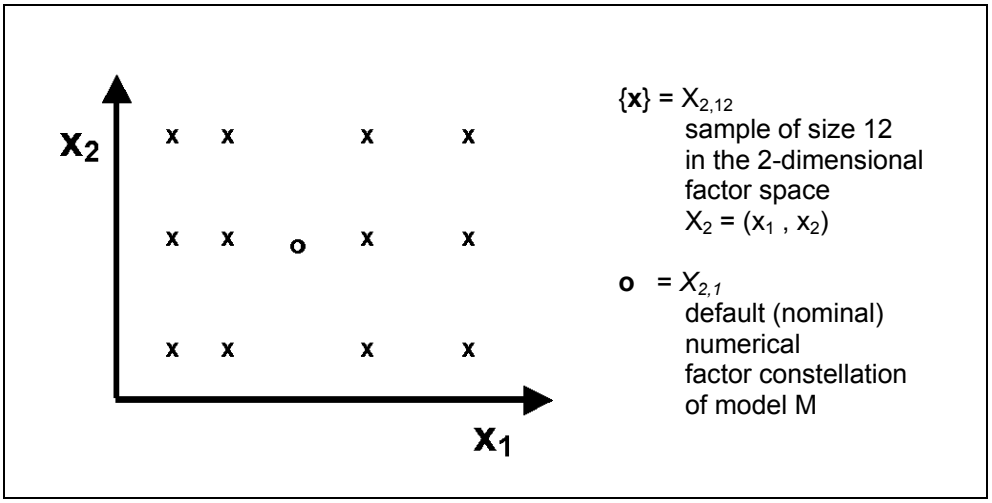


Fig. 4.5 *Sample for DFD*

SimEnv sampling strategy for DFD 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. 4.6 describes a Deterministic Factorial Design 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.

The middle scheme (b) represents a parallel scanning of these two factors at the X_2 diagonal, resulting in 1+1+1+1 = 4 model runs.

The right scheme (c) shows a complex scanning strategy of the 3-dimensional factor space $X_3 = (p_1 , p_2 , p_3)$,
resulting in (1+1+1+1)*3 = 12 model runs.

Each filled cross x in Fig. 4.6 represents a sample point in the factor space and finally a single model run of the experiment.

Example 4.1 *DFD examples for scanning multi-dimensional factor spaces*

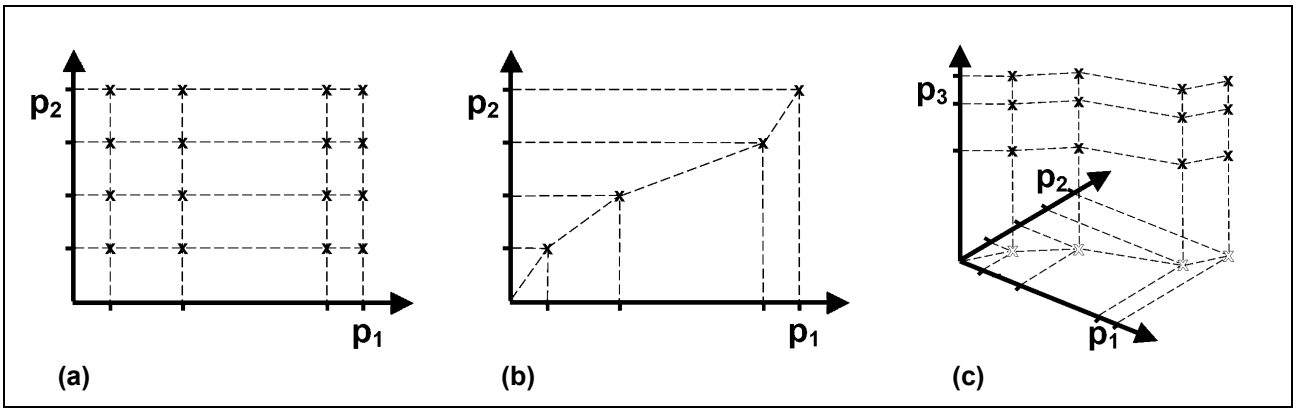


Fig. 4.6 *DFD examples for scanning multi-dimensional factor spaces*

4.5 Uncertainty Analysis – Monte Carlo Method UNC_MC

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

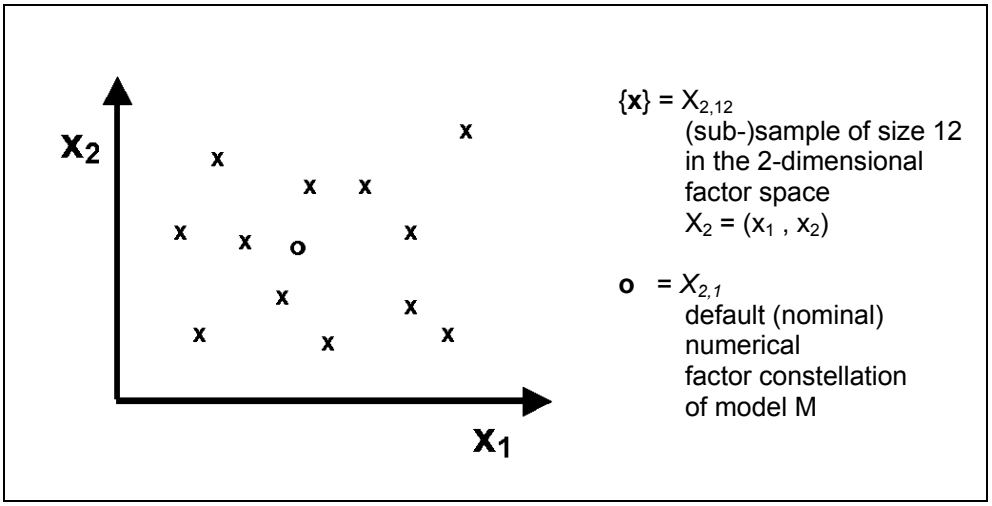


Fig. 4.7 *Sample for UNC_MC*

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

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

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 marginal (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 ("Monte Carlo runs") where the numerical factor values all of k factors x_{ij} of x_i ($1 \le i \le k$, $1 \le j \le N$) are sampled according to the marginal probability density function pdf_i.
On the basis of these assumptions, the statistical measures in Tab. 4.3 can be computed during performance of a post-processing session from an UNC_MC experiment.

Tab. 4.3 *Statistical measures for an UNC_MC experiment*
N = number of Monte Carlo runs,
z_1 ,..., z_N, $z1_1$,..., $z1_N$, $z2_1$,..., $z2_N$ are realizations of state variables z, z1 and z2. resp.
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	$\mathrm{sum}(z) = \Sigma\, z_i$
arithmetic mean	$\mathrm{avg}(z) = \Sigma\, z_i / N$
variance	$\mathrm{var}(z) = \Sigma\, (z_i - \mathrm{avg}(z))^2 / (N - 1)$
skewness	$\mathrm{skw}(z) = \Sigma\, (z_i - \mathrm{avg}(z))^3 / (N * \mathrm{var}(z)^{3/2})$
kurtosis	$\mathrm{krt}(z) = \Sigma\, (z_i - \mathrm{avg}(z))^4 / (N * \mathrm{var}(z)^2) - 3$
range	$\mathrm{rng}(z) = \max(z) - \min(z)$
geometric mean	$\mathrm{avgg}(z) = (\Pi\, z_i)^{1/N}$
harmonic mean	$\mathrm{agvh}(z) = N / \Sigma(1 / z_i)$
weighted mean (w)	$\mathrm{avgw}(z) = \Sigma\, z_i * w_i / \Sigma\, w_i$ w : weight
correlation	$\mathrm{cor}(z1,z2) = \Sigma\, (z1_i - \mathrm{avg}(z1)) * (z2_i - \mathrm{avg}(z2)) / \sqrt{\Sigma\, (z1_i - \mathrm{avg}(z1))^2 * \Sigma\, (z2_i - \mathrm{avg}(z2))^2}$
covariance	$\mathrm{cov}(z1,z2) = \Sigma\, (z1_i - \mathrm{avg}(z1)) * (z2_i - \mathrm{avg}(z2)) / (N - 1)$
linear regression coefficient to forecast z2 from z1	$\mathrm{reg}(z1,z2) = (\Sigma\, (z1_i - \mathrm{avg}(z1)) * (z2_i - \mathrm{avg}(z2))) / (\Sigma\, (z1_i - \mathrm{avg}(z1))^2)$ It is: $z2 = \mathrm{reg}(z1,z2) * z1 + \mathrm{avg}(z2) - \mathrm{reg}(z1,z2) * \mathrm{avg}(z1) + \mathrm{error}$
median	$\mathrm{med}(z)$ = middle value from increasingly ordered $\{ z_i \}$ (N = odd) mean of the two middle values from $\{ z_i \}$ (N = even)
quantile (p)	$\mathrm{qnt}^{(p)}(z)$ = that value from increasingly ordered $\{ z_i \}$ which corresponds to a cumulative frequency of N*p/100 $\mathrm{qnt}^{(50)}(z) = \mathrm{med}(z)$
confidence interval boundaries (α)	$\mathrm{cnf}^{(\alpha)}(z) = \mathrm{avg}(z) \pm t_{\alpha,N-1} \sqrt{\mathrm{var}(z) / N}$ α : probability of error $t_{\alpha,N}$: significance boundaries of Student distribution
heuristic probability density function	$\mathrm{hgr}^{(\mathrm{class})}(z)$ = number of z_i with $\mathrm{class}_{min} \le z_i < \mathrm{class}_{max}$ class_{min}, class_{max} : boundaries of equidistant classes

Tab. 4.4 summarizes those 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 ∈ [a,b] pdf(x) = 0 otherwise	a lower bound b upper bound > a it is: mean = (a+b) / 2 standard deviation = $\sqrt{(b-a)^2 / 12}$
normal	N(μ,σ^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(μ,σ^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) ~ N(μ,σ^2)
exponential	E(μ)	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 an UNC_MC experiment 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

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

and checks it against the mean avg(z).

4.6 Local Sensitivity Analysis LSA

Local sensitivity analysis LSA uses a deterministic sampling strategy 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 constellation $X_{k,1}$ and each positive 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. 4.5.

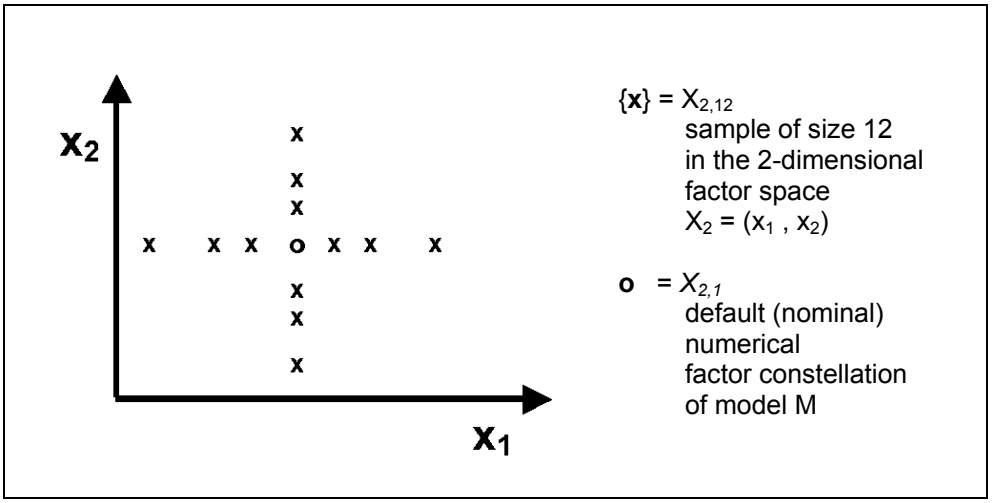


Fig. 4.8 *Sample for LSA*

Tab. 4.5 *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	$sens\_abs(z,\pm\varepsilon) = \frac{z(x \pm \varepsilon) - z(x)}{\pm \varepsilon}$	$sens\_rel(z,\pm\varepsilon) = sens\_abs(z,\pm\varepsilon) \frac{x}{z(x)}$
linearity measure	$lin\_abs(z,\varepsilon) = \frac{(z(x+\varepsilon) - z(x)) + (z(x-\varepsilon) - z(x))}{\varepsilon}$	$lin\_rel(z,\varepsilon) = lin\_abs(z,\varepsilon) \frac{x}{z(x)}$
symmetry measure	$sym\_abs(z,\varepsilon) = \frac{z(x+\varepsilon) - z(x-\varepsilon)}{\varepsilon}$	$sym\_rel(z,\varepsilon) = sym\_abs(z,\varepsilon) \frac{x}{z(x)}$

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 neighbourhood ε of a factor value t. That is why the conclusions drawn from a LSA experiment 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 neighbourhood 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 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. 4.5 additional measures can be derived on demand, e.g., abs(sym_abs(z, ε)).

A LSA 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 (nominal) values of the factors. Local sensitivity functions are calculated during experiment post-processing.

4.7 Bayesian Technique – Bayesian Calibration BAY_BC

Bayesian calibration is a method to reduce uncertainty about model factors using Bayesian techniques. The basic approach is to qualify given marginal prior probability density function pdf(X_k) of the factors X_k by deriving a representative sample. This is achieved by taking into account a vector of individual measurement values represented by their averages avg_d and their accompanied measurement errors expressed as variances var_d. They are measured from the real system that correspond to individual elements of state variables / model output z(X_k) / results from the model. For the meaning of pdf, X_k and z check Section 4.5.

Bayes' theorem

$$pdf_{po}(\text{prior} \mid avg_d, var_d) = pdf_{pr}(\text{prior}) * L(avg_d, var_d \mid \text{prior}) / pdf_c(avg_d, var_d)$$

is applied, with

$pdf_{po}(\text{prior} \mid avg_d, var_d)$	posterior distribution given data (avg_d, var_d)
$pdf_{pr}(\text{prior})$	prior distribution
$L(avg_d, var_d \mid \text{prior})$	likelihood function of the data (avg_d, var_d) given model output z(X_k)
$pdf_c(avg_d, var_d)$	normalization constant.

If the data avg_d have a low measurement error var_d then the posterior distribution is expected to represent less uncertainty than the prior distribution. Since Bayes' theorem in general can not be solved analytically for a simulation model a representative sample for the posterior distribution is generated by a Markov chain Monte Carlo method MCMC. In SimEnv the Metropolis-Hastings algorithm is used for MCMC and the argumentation and implementation in van Oijen (2008) is followed. The Metropolis-Hastings algorithm itself performs a random walk through the factor space. A chain of visited points is generated where a candidate for the posterior distribution / for the chain is generated randomly and is accepted or rejected dependent on the Metropolis ratio. The candidate point in the factor space is found by a multivariate normal jump from the previously accepted point in the chain.

The individual probability density function used for the likelihood function is determined by the distribution of the measurement errors. Here the common assumptions are made that the measurement errors are nor-

mally distributed, uncorrelated and additive. Consequently, the likelihood function follows a normal distribution $L \sim N(avg_d , var_d)$.

The Metropolis ratio

MR = post_prob (candidate point) / post_prob (prev. accepted point)

with the posterior probability

$post\_prob (\bullet) = pdf_{pr} (\bullet) * L (avg_d , var_d | \bullet)$

is used to decide whether a proposed candidate point in the factor space is accepted or not:

MR > 1	if and only if the candidate point has a higher posterior probability than the previously accepted point: candidate point is accepted
MR ≤ 1	else: candidate point is accepted with probability MR

The multivariate normal jump can result in a candidate point that is located outside the domain of definition of the prior distribution $pdf_{pr}(X_k)$ and consequently has a prior probability of zero. This may be the case for distributions with hard coded lower and/or upper bounds, e.g., the uniform distribution. While for an outranged candidate point a single simulation run will not be performed such a candidate does not contribute to the inspection of the factor space for a given chain length but reduces the efficiency of the algorithm in terms of the ratio between outranged candidates and the chain length. That's why, SimEnv offers beside the standard Metropolis-Hastings algorithm a modified version. The so-called Metropolis-with-Reflection algorithm (van Oijen, 2008) can reduce drastically the number of outranged candidates that are not subject to the Metropolis ratio MR. The algorithm tries to "reflect back" a jump outside the definition domain again within the distribution bounds.

The chain can be stopped when it has "converged", i.e., it has explored the parameter space adequately. Visual analysis of convergence can be assisted by trace plots where the value of a sampled factor is plotted versus the run number (see Fig. 4.9). Nevertheless, in SimEnv the chain length has to be specified as a fixed value before the experiment and is not subject to any stopping criterion. For more information on MCMC trace plots check for example SAS (2012).

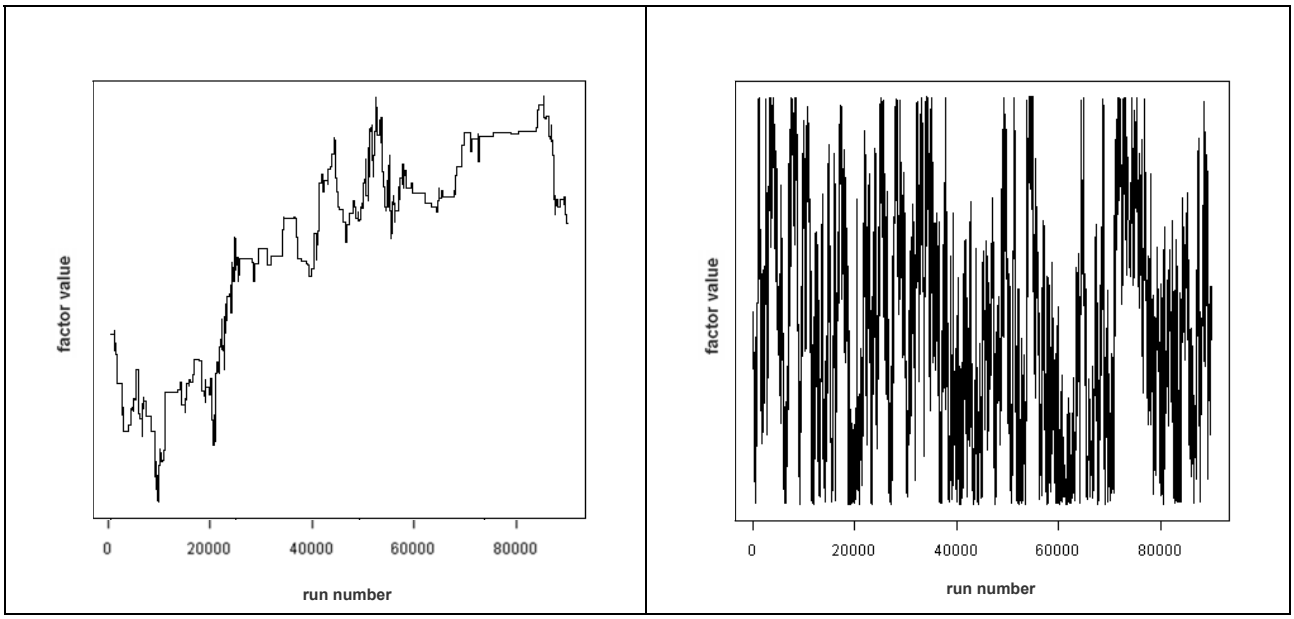


Fig. 4.9 *Trace plots of a MCMC chain for one factor*
Chain length: 100000 runs; left: chain does not converge, right: chain converges

As for the experiment type UNC_MC the prior distribution is assumed to be decomposed into marginal distributions $pdf_{pr}(X_k)$ for k all individual factors

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

Covariances between the individual factors can be specified to determine the direction of the multivariate normal jump. The Metropolis-with-Reflection algorithm does not allow for inter-factor nonzero covariances as normally they contradict the requirement in the Metropolis-Hastings algorithm that a jump in the factor space from a point p_1 to a point p_2 has the same probability as from p_2 to p_1.

Furthermore, SimEnv enables to compute the likelihood function and finally the posterior probability for different levels of complexity from data files df_i composed from a number of data points avg_d and var_d that correspond to individual elements of model state variables / output variables / results y_i:

Compose the posterior probability with a likelihood function

- for the **single likelihood function case**
  from one data file and its corresponding result: df_1 with y_1
- for the **multiple likelihood function case**
  from l > 1 data files each corresponding to its result: df_1 with y_1 ,…, df_l with y_l
  (sequence of single likelihood functions)
- for the **multiple setting likelihood function case**
  from performing each single run for m > 1 settings and
  for each setting from l ≥ 1 setting-specific data files
  each corresponding to its setting-independent result: df_1^{s1} with y_1 ,…, df_l^{s1} with y_l
  …
  df_1^{sm} with y_1 ,…, df_l^{sm} with y_l
  (sequence of multiple likelihood functions)

Example:

Single likelihood function case:	a forest growth model is applied for one patch and one likelihood function for the tree diameter is used
Multiple likelihood function case:	a forest growth model is applied for one patch and likelihood functions for tree diameter and tree height are used
Multiple setting likelihood function case:	a forest growth model is applied for several patches and one or several likelihood functions are used

The resulting likelihood function is always the product of the individual likelihood functions. The measurement error expressed as the variance determines the influence of the individual likelihood functions: The smaller the measurement error the more influential the individual likelihood function and finally the individual posterior probability.
Settings for the multiple setting likelihood function case may differ e.g. in model calibrations or site conditions. Such an experiment design is useful for identifying an overall posterior sample rather than individual setting-related samples.

4.8 Optimization – Simulated Annealing OPT_SA

The optimization experiment OPT_SA in SimEnv uses a stochastic strategy to sample X_k. It is an experiment type where the sample is generated during experiment performance and not at experiment preparation. The general approach of any single-objective optimization method 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 $X_k = (x_1 ,..., x_k)$:

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

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 6.7 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 Markov Chain Monte Carlo MCMC method (check also Section 4.7) for examining the state equations of n-body systems. The concept is based on the manner in which metals recrystalise 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. The feature of accepting by a certain probability a state with a higher energy than the previous state allows for "climbing" out from local sub-optimal energy states during simulated annealing.

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 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 should "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.

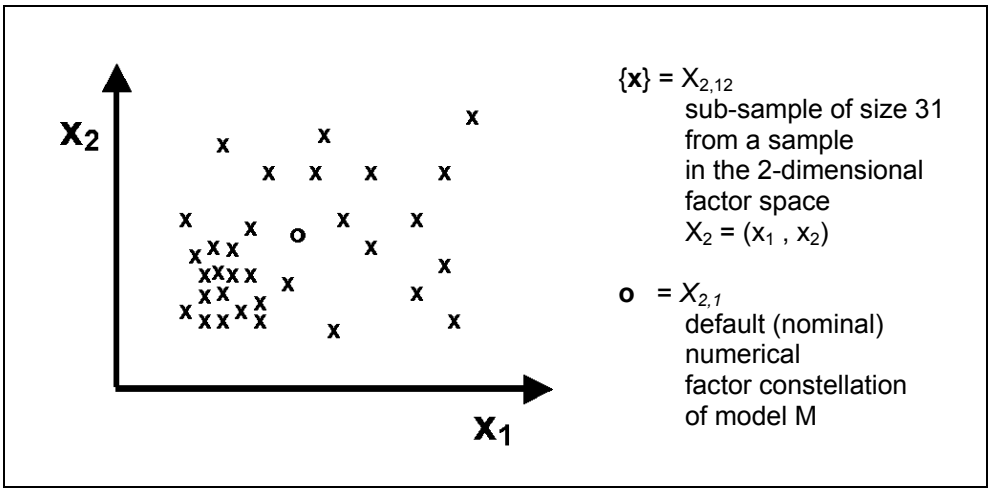


Fig. 4.10 *Part of a sample for OPT_SA, generated during the experiment*

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 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 the transfer of sampled numerical values of model factors under investigation from the simulation environment to the model and to the transfer of 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

The 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 source code entities (parameters, initial values, drivers). When performing the model these entities are modified numerically by the sampled values and the default (nominal) values of the factors according to the specified factor adjustment types. The process of such a modification is called an **adjustment**. The factor adjustment type specifies how to interfere the current sampled value with the default (nominal) 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.

Implementation 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,

- each experiment factor and
- each model output variable

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

Tab. 5.1 *Generic SimEnv model 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 / 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 an entity (in general, an array / a state variable) of the model with a model output variable from <model>.mdf and output the entity to SimEnv data structures
simenv_slice_<lng>	enable slicing, i.e., a repetitively partial output of a model output entity.
simenv_end_<lng>	close model coupling interface

With the function simenv_slice_<lng> a slice / portion an output entity / array / variable can be ouput to SimEnv data structures by a subsequent function call of simenv_put_<lng>. In particular, such slicing can be applied formodels with multi-dimensional output 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. The simenv_slice_<lng>-function ensures that model output over the omitted source code dimension can be handled in experiment post-processing in common.

Example:

A typical example is a model to study the temporal dynamics of a process on a latitude-longitude grid by analyzing after the experiment a variable out_var(lat,lon,time).

If in the model source code the corresponding array mod_var(lat,lon) only depends on lat and lon since the temporal dynamics is modeled in place then this mismatch is handled by the SimEnv interface slice function.

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

The alignment of the contents of the SimEnv model output and experiment description file 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 paradigm is implemented in a flexible manner:

- **Entry in SimEnv description file missed and corresponding function call in model source code available:**
  A function call in the source code where an experiment factor from <model>.edf and/or a model output variable from <model>.mdf is not associated with is handled during the model performance in such a way that the factor is unadjusted and/or the model output variable is not output. In particular, the value of a factor remains the value it had in the model source code before calling the simenv_get_* function call.
  The factor and/or model output variable is not stored in SimEnv experiment output data structures and can not be addressed in SimEnv experiment post-processing.
  This approach 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 a special experiment and/or requested for model output.

- **Entry in SimEnv description file available and corresponding function call in model source code missed and/or unperformed:**
  A model entity that is defined in the corresponding experiment description file <model>.edf as a factor and/or in the model output description file <model>.mdf as an output variable and where the corresponding SimEnv functions in the model source code are missing can be addressed during experiment post-processing. They are completely undefined.
  For IEEE SimEnv model output such a factor and/or model output variable is set to its corresponding nodata value. For NetCDF output the factor and/or model output variable is not available from / undefined in the NetCDF file(s).

A regular matching between the entity in a model output description file and the used SimEnv interface function in the model source code as well as the above exceptions are reported to the interface log file <model>.mlog (cf. Tab. 11.8).

Native model output does not influence performance of the model in SimEnv and there is no necessity to disable this output when the model is interfaced to SimEnv. The user only has to ensure that for an experiment control by the Distributed Resource Manager DRM 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 subdirectory (cf. Example 15.10). user model native output to the terminal is redirected during the experiment to the SimEnv experiment log file <model>.nlog.

For running an interfaced model outside SimEnv it comes with a dummy library to link / run the model with. Its corresponding interface functions 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 level and for ASCII files. Mixed language models as well as distributed models (cf. Section 5.11) can be interfaced to SimEnv.

For multi-dimensional model output the simenv_put_<lng> function expects the output stored in a model source code array (the “entity” as described in Tab. 5.1 above) according to the order model that is used by the corresponding language <lng>. The sequence of the dimensions of the source code array is determined by the sequence of the coordinates for the associated variable in the file <model>.mdf (check Section 5.2).

Tab. 5.2 *Language suffices and multi-dimensional array order models for SimEnv model interface functions*
(for the Mathematica and GAMS interface check Section 5.6 and 5.7, resp.)

Model source code	<lng>	Check Section	Order model for simenv_put_<lng> (cf. Section 15.7 – Glossary)
ASCII file	as	5.9	column-major
C/C++	c	5.4	row-major
Fortran	f	5.4	column-major
Java	ja	5.5	row-major
Matlab	m	5.5	column-major
Python	py	5.5	row-major
Shell script	sh	5.8	simenv_put_sh not available – check Section 5.8

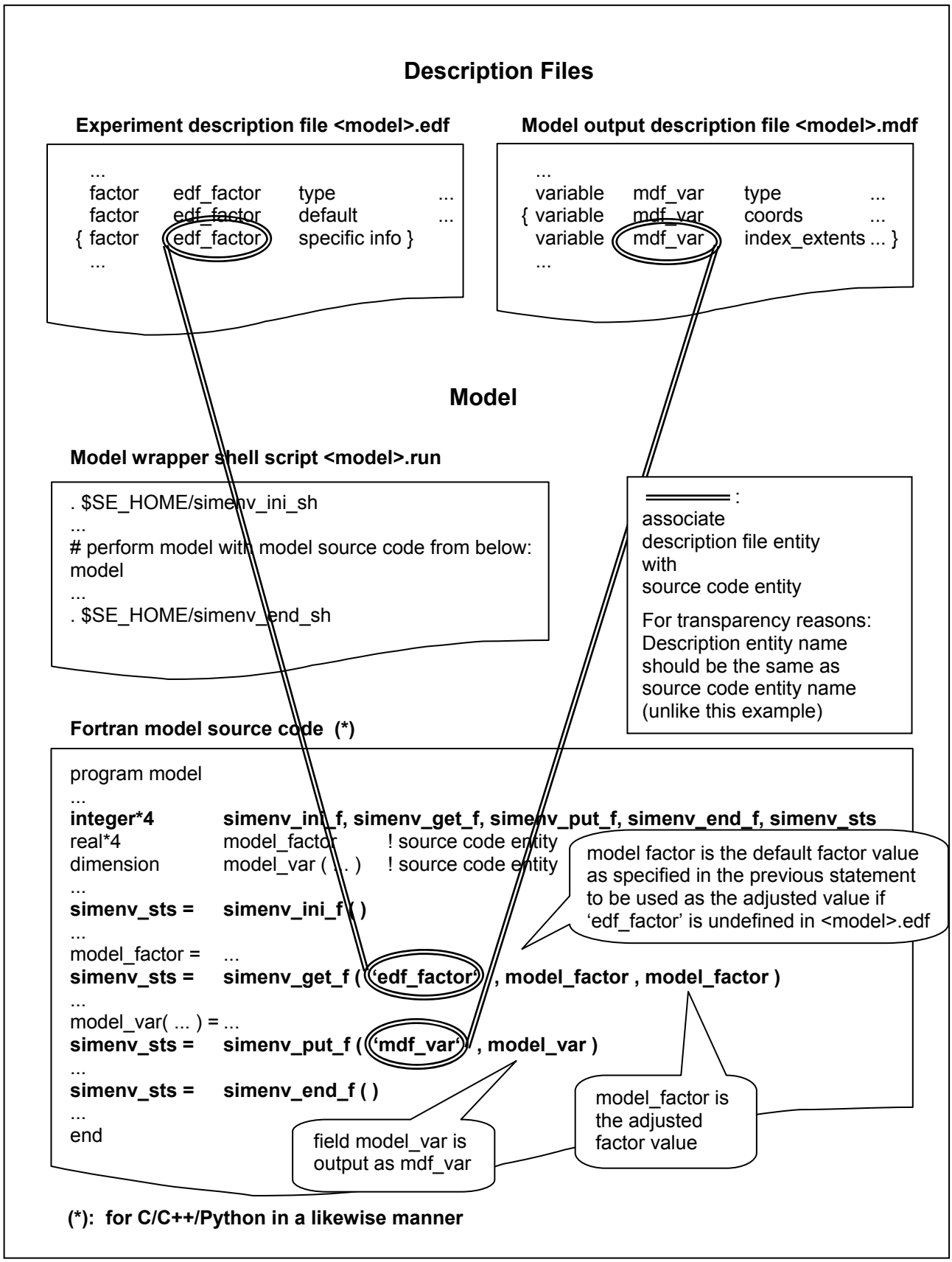


Fig. 5.1 *Conceptual scheme of the model interface for C/C++, Fortran, Python, Java and Matlab*

5.2 Coordinate and Grid Assignments to Variables

To each model output variable that is to be stored during the perfomance of a SimEnv experiment to SimEnv output data structures

- A dimensionality **dim** **> 0**
- Extents **ext(i)** **> 1** with i=1 ,..., dim
- Coordinates **coord(i)** with i=1 ,..., dim

are assigned to.

The **dimensionality** is the number of dimensions the model output variable has. A dimensionality of 0 corrsponds to a single scalar, dimensionality of 1 (2) corresponds to to one (two) dimension(s) and the values of the variable are represented by a vector (a matrix). In general, a dimensionality of n corresponds to an n-dimensional array.

The **extent** relates to each dimension of a model output variable and represents the number of elements in that dimension. Extents are always greater than 1. A 4x5 matrix $(mat_{ij})_{i=1,...,4 \;\; j=1,...,5}$ has an extent of 4 for the first dimension and of 5 for the second one.

To each dimension a **coordinate** is assigned to. Coordinates have a name and a well defined number of coordinate values. A subset (or even the whole set) of values of a coordinate is assigned to the dimension of that variable. Consequently, the number of coordinate values is the extent of this dimension. A single model output variable must not have identical coordinate names but can have identical coordinate values.

Finally, coordinate axes are defined. A **coordinate axis** is a strictly monotonic ordered sequence of coordinate values and has a description and a unit. A complete coordinate axis or a part of a coordinate axis is assigned to each dimension of a model output variable. Consequently, each variable 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 and block-structured. Rectlinear refers to an orthogonal / Cartesian coordinate system and block-structured to variable distances between adjacent coordinate values. A model output variable values then exist on the rectilinear grid, spanned up from the coordinate values of the coordinate axes. The left side of Fig. 5.2 shows a model output variable of dimensionality 2. The first dimension has an extent of 5 and is associated with coordinate axis $coord_1$.
Variables of dimensionality 0 do not have extents and coordinate / coordinate axes / coordinate system assignments.

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.

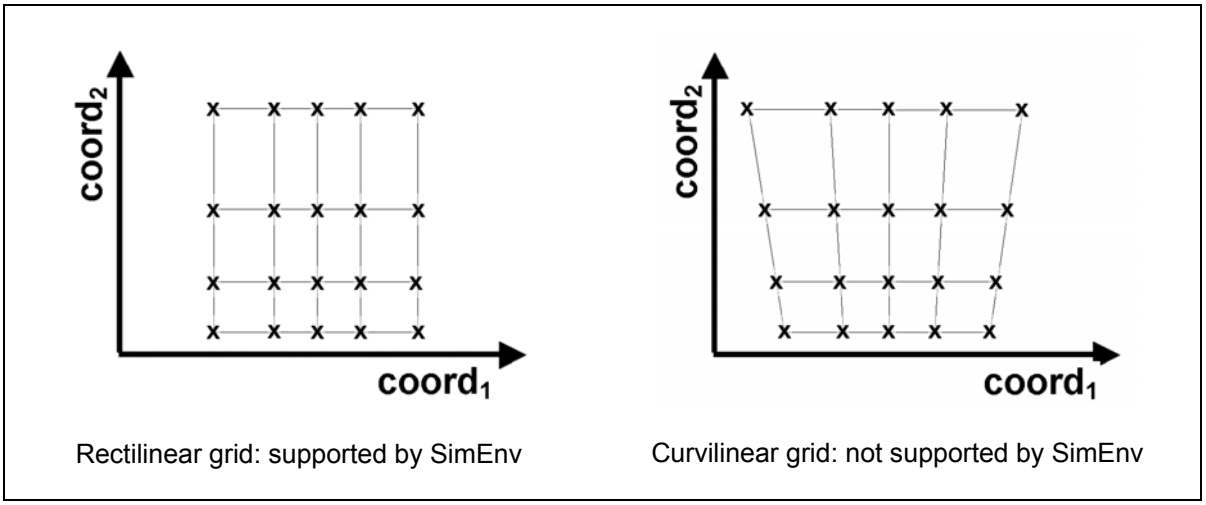


Fig. 5.2 *Grid types*

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 with a dimensionality > 0 is always 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 (line type: m = mandatory, o= optional)*

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. 12.5)
variable	<var_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	see below	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	see below	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	see below	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.

It is always a good idea that the model output variable name correspond to the name of the variable in the simulation model code. Association between the 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 12.1 on page 203 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

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

- Coordinate names <co_name> and variable names <var_name> must differ from factor names <factor_name> 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 used in 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. Consequently, the sub-keyword ***coords*** is mandatory for variables with dimensionality > 0 and undefined for variables with dimensionality 0.
- 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

  This sub-keyword is optional for variables with dimensionality > 0 and undefined for variables with dimensionality 0.
- 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.

  This sub-keyword is mandatory for variables with dimensionality > 0 and undefined for variables with dimensionality 0.
- 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.
- The model output description as specified by “general descr <string>” is used for NetCDF model and/or post-processor output files (check Section 10.1.1).

Tab. 5.4 *SimEnv data types*

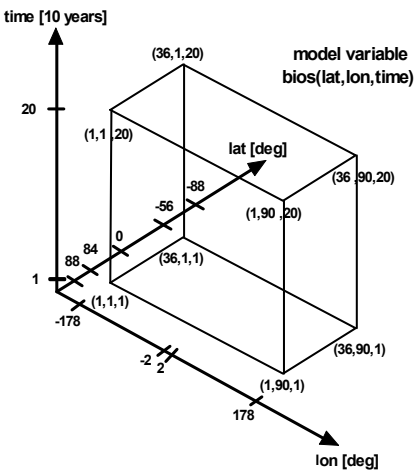
SimEnv data type (synonyms)		Description		Restriction
byte	int*1	1 byte	integer	not for Python and Java models
short	int*2	2 bytes	integer	not for Python and Java models
int	int*4	4 bytes	integer	
float	real*4	4 bytes	real	
double	real*8	8 bytes	real	not for Python and Java models

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 7:

```
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            atmo_unit
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            bios_unit
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 files: 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 (nominal) 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 = 1 warning: function multiple called: only the first call is executed. = 2 I/O error for experiment output file = 3 error memory allocation = 4 I/O error for simenv_edf_bin.tmp as the structured representation of <model>.edf = 5 I/O error for simenv_mdf_bin.tmp as the structured representation of <model>.mdf = 6 I/O error for <model>.smp = 7 wrong single run number

Function name	Function description	Arguments / function value	Argument / function value description
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 experiment output file
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]	return code = 0 ok = 1 var_name undefined

Function name	Function description	Arguments / function value	Argument / function value description
		(function value)	= 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 experiment output file

To Tab. 5.5 the following remarks apply:

- 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.
- For multi-dimensional model output the simenv_put_f function expects the output stored in the Fortran model source code array (the “field” as described in Tab. 5.5) according to the column-major order model (cf. Section 15.7 – Glossary) that is used by Fortran by default. For C/C++ and simenv_put_c the row-major order model (cf. Section 15.7 – Glossary) used by C/C++ by default is expected. The sequence of the dimensions of the source code array is determined by the sequence of the coordinates for the associated variable in the file <model>.mdf (check Section 5.2).
- 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. Section 11.8).
- Application of simenv_slice_[f | c] demands a corresponding slice entry in the configuration file <model>.cfg. For more information check Section 11.1.
- 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 and/or C/C++.
- Apply the shell script
  link_simenv_mod_[f | c | cpp].sh <model_name>
  from the SimEnv library directory $SE_HOME/lib to compile and link an interfaced Fortran / C / C++ 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. 15.13 lists the additionally used symbols when interfacing a Fortran or C/C++ model to SimEnv.
- In
  - Example 15.1 on page 224 the model world_f.f
  - Example 15.3 on page 227 the model world_c.c
  - Example 15.4 on page 229 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 occurred

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 occurred
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

To Tab. 5.6 the following remarks apply:

- 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.
- For multi-dimensional model output the simenv_put_m function expects the output stored in a model source code array (the “field” as described in Tab. 5.6) according to the column-major order model (cf. Section 15.7 – Glossary) that is used by Fortran by default. For Python / Java and simenv_put_py / simenv_put_ja the row-major order model (cf. Section 15.7 – Glossary) used by Python / Java by default is expected. The sequence of the dimensions of the source code array is determined by the sequence of the coordinates for the associated variable in the file <model>.mdf (check Section 5.2).
- 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. Section 11.8).
- Application of simenv_slice_[py | ja | m] demands a corresponding slice entry in the configuration file <model>.cfg. For more information check Section 11.1.
- 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.
- 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_ja
- 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 11.7.
- 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 11.9.

In Example 15.5 on page 230 the Python model world_py.py is described in detail, in Example 15.6 on page 231 the Java model world_ja.java and in Example 15.7 on page 232 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 99) 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 106) <model>.ini has to be performed again. To force this specify in <model>.cfg (cf. Section 11.1 on page 181) 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 11.9.

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 (nominal) 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 those 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 by 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 source code at such a position where all the GAMS variables are declared. Directly before the include statement the factor default (nominal) 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 source code 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 or absolute paths.

In Example 15.9 on page 235 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 source code 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 11.9.

5.7.1 Standard Dot Scripts for GAMS Models

<model>.ini
<model>.ini (cf. Section 7.1 on page 99) 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 106) <model>.ini has to be performed again. To force this, specify in <model>.cfg (cf. Section 11.1 on page 181) for the sub-keyword *restart_ini* the value “yes”.

<model>.run
<model>.run (cf. Section 7.1 on page 99) 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 99) 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

For each GAMS model an ASCII GAMS description file **<model>.gdf** has to be supplied It is intended 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 12.1 on page 203 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

Tab. 5.7 *Elements of a GAMS description file <model>.gdf (line type: m = mandatory, o = optional)*

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. 12.5)
		time_limit	o	1	<val_int>	wall clock time 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
		script	o	1	{<directory>/} <file_name>	shell script to be performed before each single GAMS model run
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:

- 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.
- If the sub-keyword ***options*** is not specified the GAMS main model is started with the default options

  ```
  ll=0 lo=2 dp=0
  ```

  complemented by

  ```
  lf=main_model<simenv_run_char>.nlog Optdir=../
  ```

  Check also Section 5.7.3.
- By the sub-keyword ***script*** a shell script can be specified that will be performed before each single GAMS model run. By default, the shell script is expected to be located in the current workspace. The shell script is performed from the sub-directory run<simenv_run_char> of the current workspace (cf. Section 5.7.3). The shell script has one argument, the current run number <simenv_run_char> formed from six characters. For example, the script can be used to copy some files into the sub-directory run<simenv_run_char> where the GAMS model is called from. Make sure by the Unix / Linux command chmod u+x {<directory>/}<file_name> that the shell script has execute permission.
- 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”.
- 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 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.
  - 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
  - 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.

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 <model>_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. 11.8, 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 performance 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 with the options

```
<values_from_sub_keyword_options>
lf=main_model<simenv_run_char>.nlog Optdir=../
```

In case the sub-keyword *options* is not specified the main model is called with

```
ll=0 lo=2
lf=main_model<simenv_run_char>.nlog 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 main model and sub-models Optdir=../ implies that all GAMS solver option files have to be located in the current working directory.

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 99) 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.

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**		

To the Tab. 5.8 following remarks apply:

- 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 the values of 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. 11.10 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 236 the model shell script world_sh.run is described in detail.

```
. $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_as 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.
$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.

Having a model output variable definition as in Example 5.1 on page 44.

`$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*

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 column-major order 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. Section 11.8).
- 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 12.3 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.

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 11.1).

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 script script <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). Normally, this shell script will call the link scripts for interfaced models in $SE_HOME/lib.
- 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 11.1) 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 Distributed Resource Manager DRM by using the llsubmit (for LoadL) or qsub (for PBS / Torque) command. Start an experiment from a login node of the compute cluster and run the experiment at the login node in foreground. SimEnv submits from the login machine all single runs to The DRM and directly finishes afterwards. DRM then takes 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 DRM services for the end of the last parallel single model run.
To support bookkeeping of SimEnv applications on PIK's compute cluster under LoadL control 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
```

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 under DRM LoadL control:
llsubmit my_parallel_model.jcf
# or run a single run of the model under PBS/Torque control:
qsub my_parallel_model.pbs

. $SE_HOME/bin/simenv_end_sh
```

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

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.

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 these 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 12.1 on page 203 with the keywords, names, sub-keywords, and value as in Tab. 6.1.

Tab. 6.1 *Elements of an experiment description file <model>.edf for all experiment types (line type: m = mandatory, o = optional)*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
general	<nil>	descr	o	any	<string>	experiment description
		type	m	1	[GSA_EE | DFD | LSA | UNC_MC | OPT_SA]	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 (nominal) value in simenv_get_* to get the resulting adjusted factor value
		default	m	1	<val_float>	factor default (nominal) value <factor_def_val>
		sample	see below	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:

- A factor name is the symbolic parameter / driver / initial 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 (nominal) value as specified in <model>.edf and not its default (nominal) value as specified in the model code is used in the model interface function simenv_get_*.**
- **The file <model>.smp always contains the sampled values according to the experiment and factor specific sampling scheme as defined by the sub-keyword *sample* without applying the adjustment.**
- **Consequently, in general the factor values in <model>.smp differ from the factor values as stored in the SimEnv experiment output. In particular, this is the case for**
  - **factor adjustment type = add and <factor_def_val> ≠ 0.**
  - **factor adjustment type = multiply and <factor_def_val> ≠ 1.**
  - **factor adjustment type = relative and any <factor_def_val>**
- For the factor adjustment types “multiply” and ”relative” a default (nominal) value <val_float> = 0. is forbidden.
- All experiment specific information is explained in the appropriate Sections.
- Specify at least one experiment factor.
- The sub-keyword ***sample*** is conditional for experiment type DFD, undefined for experiment type LSA and mandatory for all other experiment types.
- 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 (nominal) 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 12.1 on page 203), 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.
- The experiment description as specified by “general descr <string>” is used for NetCDF model and/or post-processor output files (check Section 10.1.1).

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 (nominal) 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          <experiment type>

factor     p1    descr         parameter p1
factor     p1    unit          p1_unit
factor     p1    type          set
factor     p1    default       1.
factor     p1    sample        <experiment type specific>

factor     p2    type          set
factor     p2    default       2.
factor     p2    sample        <experiment type specific>

specific         <experiment type specific>
```

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

6.1.1 Elements of <model>.edf for Experiment Types with Probabilistic Sampling

For experiments with probabilistic sampling (GSA_EE, GSA_VB, UNC_MC and BAY_BC, but not OPT_SA) the additional sub-keywords are available as defined in Tab. 6.3.

Tab. 6.3 *Additional elements of <model>.edf for experiment types with probabilistic sampling (line type: m = mandatory, o = optional)*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
factor	<factor_ name>	sample	m	1	[distr <distribution> | file {<directory>/} <file_name>]	distr: distribution and distribution parameters to derive a sample of values <factor_smp_val> file: file name to import an external sample of values <factor_smp_val>
		sample_ method	see below	1	[pseudo | stratified | quasi]	random sampling method (only for "distr <distribution>")
		sample_ include	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{n1}$>: <real_val$_{n2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to include for sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
		sample_ exclude	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{m1}$>: <real_val$_{m2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to exclude from sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
specific	<nil>	seed	o	1	[random | fixed]	method how to initialize the random number generator

Be careful when specifying for an experiment with probabilistic sampling 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 shifted by the specified factor default (nominal) value <factor_def_val>. For the factor adjustment types "multiply" and "relative" the specified distribution will be adulterated normally by the factor default (nominal) value <factor_def_val>.

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

- Sub-keyword ***sample***:
  - distr <distribution>:
    For implicitly specified distributions according to Tab. 6.4 sample values <factor_smp_val> are generated from the distribution with the assigned distribution parameters.
    For <distribution> = <distr_shortcut> (<distr_param_1> { , <distr_param_2> }) check Tab. 6.4.
  - file {<directory>/}<file_name>:
    For explicitly specified samples by an ASCII file {<directory>/}<file_name> the sample values of any distribution are taken directly from this file. Each record of the ASCII file can hold only

one sample value with an exception for experiment type GSA_VB. For GSA_VB two values are expected: the first for the sample and second for the re-sample. For the other syntax rules for ASCII data files check Section 12.3. Sample size has to be identical to <val_int> from the sub-keyword *runs*. Explicitly specified samples can not be applied for experiment types GSA_EE and BAY_BC.

- Sub-keyword ***sample_method***:
  The sub-keyword is mandatory for all experiment types with random sampling and for *sample* = "distr …". For sample = "file …" the sub-key-word is undefined. A sample method can not be specified for experiment types GSA_EE and BAY_BC.
  The sub-keyword *sample_method* allows to select the random sampling method to be applied individually for each factor. For **"pseudo"** random numbers a deterministic algorithm is used that approximates the properties of the given factor's distribution by a sequence of generated numbers. The sequence depends on an initial values that is controlled by the sub-keyword *seed*. For **"stratified"** random sampling the Latin Hypercube approach (e.g., McKay *et al.*, 1979, Iman & Helton,1998, Helton & Davis, 2000) is used. The sampling range of the factor as determined by its distribution is divided into <val_int> (from the sub-keyword *runs*) intervals of equal probability according to the given distribution. From each interval exactly one sampling point is drawn. The sampling point within each interval is drawn with a pseudo random number approach. The **"quasi"** random sampling is based on the usage of pre-defined Sobol' sequences (Sobol', 1967 & 1976) for the interval <0,1>. Sobol' quasi random sampling is also known as LP_τ sampling. In general, quasi random sampling results in a low-discrepancy sequence which show a regular distribution pattern in the factor space and hence qualifies it for numerical integration. For quasi random sampling the first values of the sequence are skipped: 1500 values for a fixed seed, for a random seed the number of values is randomly selected within the interval <1000,2000>. Check also the Figures below.
  Pseudo and stratified random sampling at first generate a factor sample uniformly or stratified uniformly distributed on <0,1>. For quasi random sampling a Sobol' sequence is taken. Afterwards, these values $x_{i<0,1>}$ are transformed to the target distribution as specified by <distribution>. The definition domain of the distribution according to *sample_include* and *sample_exclude* sub-keywords is taken into account and random numbers of the target distribution are identified as the quantiles of $x_{i<0,1>}$.
- Sub-keywords ***sample_include*** and ***sample_exclude***:
  With the sub-keywords *sample_include* and *sample_exclude* intervals to sample from can be specified explicitly when the distribution is defined by "sample distr <distribution>". A typical example for application of these two sub-keywords is to exclude values from the distribution tails. If include or exclude intervals are not specified the corresponding overall defaults from the distribution are applied. Use +inf to declare a positive infinite interval bound and -inf for a negative infinite interval bound. The result from the overlay of all include and exclude intervals with the definition domain of the distribution for probabilities > 0 can be several non-connected intervals to sample from. Sample_include and sample_exclude can not be used for distributions imported from files.
  Keep in mind that the generated sample of factor values normally does no longer follow the statistical properties of the specified distribution when applying sample_include and/or sample_exclude. In particular, moments will differ from those as determined by the distribution and its parameters.
- Sub-keyword ***seed***:
  This sub-keyword controls the seed of the used random number generator for implicitly specified distributions ("distr <distribution>"). A random seed will result in different generated samples for repetitive performance of the same experiment while with a fixed seed the generated samples will be identical also across different machines (without taking into account numerical inconsistencies, e.g., rounding errors). If the sub-keyword is not stated initial seed is set to random.

Tab. 6.4 *Probability density functions and their parameters*

Distribution function	distr_ shortcut	distr_param_1	distr_param_2	Restriction
uniform	U	lower bound	upper bound	lower bound < upper bound
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 4.5 and Tab. 4.4.

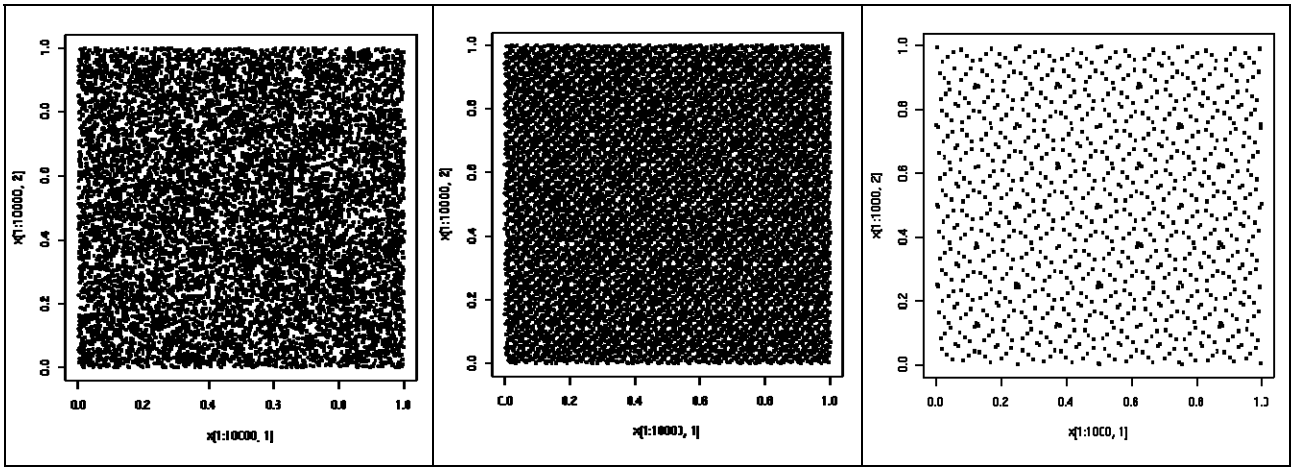
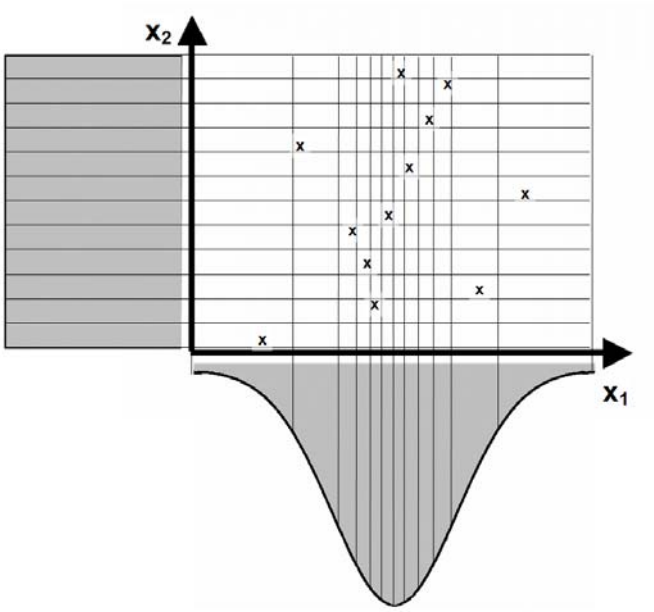


Fig. 6.1 *Probabilistic sampling: Pseudo and quasi sampling*
left: pseudo sampling (sample size N=10000)
mid: quasi sampling (N=10000),
right: quasi sampling (N=1000).
Factors on abscissa and ordinate ~ U(0,1).
(From http://en.wikipedia.org, GNU Free Publication License)



Sample size
N = 12 single simulation runs.

Factor x_1 is normally distributed
Factor x_2 is uniformly distributed.

Fig. 6.2 *Probabilistic sampling: Latin hypercube sampling*

```
# only sampling relevant sub-keywords (without sample_method) are shown:

factor     p1    descr             sample without restrictions in
                                   (-inf,+inf)
factor     p1    sample            distr N(0,0.04)

factor     p2    descr             as p1 but sample only within the
                                   3-sigma range
factor     p2    sample            distr N(0,0.04)
factor     p2    sample_exclude    -inf:-0.6 , 0.6:+inf

factor     p3    descr             equivalent to p2
factor     p3    sample            distr N(0,0.04)
factor     p3    sample_include    -0.6:0.6

factor     p4    descr             as p2 but exclude additionally (.10,.15)
factor     p4    sample            distr N(0,0.04)
factor     p4    sample_exclude    -inf:-0.6 , 0.6:+inf,.10:.15

factor     p5    descr             equivalent to p4
factor     p5    sample            distr N(0,0.04)
factor     p5    sample_include    -0.6:0.6
factor     p5    sample_exclude    .10:.15

factor     p6    descr             sample from (.5,+inf)
factor     p6    sample            distr L(0,0.04)
factor     p6    sample_exclude    xxx:.5        xxx = any value ≤ 0.
```

Example 6.2 *Include / exclude for probabilistic sampling in an experiment description file <model>.edf*

6.2 Global Sensitivity Analysis – Elementary Effects Method GSA_EE

The experiment specific information for experiment description files in Tab. 6.1 on page 69 is defined for GSA_EE experiments as follows:

Tab. 6.5 *Experiment specific elements of an edf file for GSA_EE (line type: m = mandatory, o = optional)*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
factor	<factor_ name>	sample	m	1	distr <distribution>	distribution and distribution parameters to grid the factor for equidistant quantiles
		sample_ include	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{n1}$>: <real_val$_{n2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to include for sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
		sample_ exclude	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{m1}$>: <real_val$_{m2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to exclude from sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
specific	<nil>	levels	m	1	<val_int>	number of levels to span up a p-level grid in the factor cube that is composed from equidistant quantiles for each factor. p ≥ 4, even
		trajectories	m	1	<val_int>	number of trajectories r to position randomly at the p-level grid r ≥ 5
		seed	o	1	[random | fixed]	method how to initialize the random number generator

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

- The **jump width** Δ in the corresponding quantile space is optimally selected as Δ = (p/2)/(p-1) (check Section 4.2)
- For a grid with **p levels** the i-th level (i=1 ,…, p) corresponds to the (1+2*(i-1)) / (2*p)-quantile. Consequently, level 1 is the 1/(2*p)-quantile and level p is the 1-1/(2*p)-quantile.
  For p=4 the quantiles are 1/8 , 3/8 , 5/8 , 7/8.
  For p=6 the quantiles are 1/12 , 3/12 , 5/12 , 7/12 , 9/12 , 11/12.
  Keep in mind that distribution tails are cut by this approach. In particular, for a factor with a uniform distribution U (lower bound , upper bound) the two bounds will not be used as factor values for the resulting grid in the factor space.

- To ensure that trajectories do not have to share grid points the ratio between the number of available grid points p^k in a k-dimensional p-level factor cube and the r*(k+1) points of r trajectories has to be greater or equal than 3: $p^k / r*(k+1) \geq 3$
- For **all the other sub-keywords** check Section 6.1.1 on page 72.

6.2.1 Special Features in GSA_EE, Run Sequence

To ensure that the r trajectories optimally scan the gridded factor space the following approach is used for generating them:

- The first trajectory is selected randomly.
- Assuming to have already determined r_{det} trajectories.
  Then generate randomly $10+5*(r-r_{det}+1)$ potential trajectories and select from them as the trajectory $r_{det}+1$ that one which has to the existing r_{det} trajectories the maximum distance in the quantile space.

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

```
loop over trajectories
        loop  over successive sampling points of the trajectory
                trajectory point
        endloop
endloop
```

6.2.2 Example

```
general          descr         Experiment description for the examples
general          descr         in the SimEnv User Guide
general          type          GSA_EE

factor     p1    descr         latitudinal phase shift
factor     p1    unit          pi/12.
factor     p1    type          set
factor     p1    default       1.
factor     p1    sample        distr U(-12,12)  use factor values -9, -3, 3, 9

factor     p2    type          set
factor     p2    default       2.
factor     p2    sample        distr U(1,10)    use factor values 2.25, 4.75, 7.25,
                                                8.75
factor     p3    type          set
factor     p3    default       3.
factor     p3    sample        distr U(-12,12)

factor     p4    type          set
factor     p4    default       4.
factor     p4    sample        distr U(1,10)

specific         levels        4                use quantiles .125, .375, .625, .875
specific         trajectories  10
```

Example file: world.edf_GSA_EE

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

6.3 Global Sensitivity Analysis – Variance-Based Method GSA_VB

The experiment specific information for experiment description files in Tab. 6.1 on page 69 is defined for GSA_VB experiments as follows:

Tab. 6.6 *Experiment specific elements of an edf file for GSA_VB*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
factor	<factor_ name>	sample	m	1	[distr <distribution> | file {<directory>/} <file_name>]	distr: distribution and distribution parameters to derive a sample of values <factor_smp_val> file: file name to import an external sample of values <factor_smp_val>
		sample_ method	see ex-plan.	1	[pseudo | quasi | stratified]	sampling strategy (only for *sample* "distr <distribution>")
		sample_ include	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{n1}$>: <real_val$_{n2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to include for sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
		sample_ exclude	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{m1}$>: <real_val$_{m2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to exclude from sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
specific	<nil>	seed	o	1	[random | fixed]	method how to initialize the random number generator
		runs	m	1	<val_int>	number of runs per factor

To Tab. 6.7 the following additional rules and explanations apply:

- For the sub-keyword ***sample*** and reading the sample and the sub-sample from a file (sample file …) each record of the ASCII file must hold exactly two sample values where the first value is for the sample and second for the re-sample. For the other syntax rules for ASCII data files check Section 12.3.
- For the sub-keyword ***run***:
  The number of resulting runs for experiment type GSA_VB is <val_int> * (k+2) where k is the number of factors to be investigated.
- For **all the other sub-keywords** check Section 6.1.1 on page 72.

6.3.1 Run Sequence

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

loop over the number of runs per factor as specified by the sub-keyword *runs* in <model>.edf.
    sample point from S
    re-sample point from R
    loop over resulting sample RS_i for factor i (i=1 ,…, k)
        resulting sample point from RS_i
    endloop
endloop

For S, R, and RS_i check Section 4.3.

6.3.2 Example

```
general       descr           Experiment description for the examples
general       descr           in the SimEnv User Guide
general       type            GSA_VB

factor   p1   type            set
factor   p1   default         1.
factor   p1   sample_method   pseudo
factor   p1   sample          distr U(-12,12)

factor   p2   type            set
factor   p2   default         2.
factor   p2   sample_method   stratified
factor   p2   sample          distr N(2,0.4)
factor   p2   sample_include  0:+inf

factor   p3   type            set
factor   p3   default         3.
factor   p3   sample_method   pseudo
factor   p3   sample          distr U(-12,12)

factor   p4   type            set
factor   p4   default         4.
factor   p4   sample_method   stratified
factor   p4   sample          distr N(4,0.4)
factor   p4   sample_include  0:+inf

specific      runs            250              in total 250*(4+2)+1 runs
```

Example file: world.edf_GSA_VB

Example 6.4 *Experiment description file <model>.edf for GSA_VB*

6.4 Deterministic Factorial Design DFD

The experiment specific information for experiment description files in Tab. 6.1 on page 69 is defined for a DFD experiment as follows:

Tab. 6.7 *Experiment specific elements of an edf file for a DFD experiment (line type: m = mandatory)*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
factor	<factor_ name>	sample	see be-low	1	<val_list>	value list of factor samples <factor_smp_val> (for syntax see Tab. 12.5)
specific	<nil>	comb	m	≥ 1	[default | <combination> | file {<directory>/} <file_name> { [strict | nonstrict] }]	information how to scan the spanned factor space

To Tab. 6.7 the following additional rules and explanations apply:

- Sub-keyword ***sample*** is either mandatory for all factors or undefined for all factors.
- 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*.

6.4.1 Formalisation of the Inspection Strategy, Run Sequence

The experiment type DFD allows for a flexible deterministic inspection strategy of the factor space in combined sub-spaces:

- The combination **<combination>** defines the way in which the space spanned by the experiment factors will be inspected by SimEnv. This is done by concatenating 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”).

    Example: compare with the experiment description file DFD_a in Example 6.5
  - **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.

    Example: compare with the experiment description file DFD_b in Example 6.5

- The operators „,“ and „*“ can be multiple used in <combination>. The operator „,“ has a higher priority than the operator „*“. Parentheses are not allowed:

  Example: compare with the experiment description file DFD_c in Example 6.5

  $x_1 * x_2$, $x_3 * x_4$ 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.

- 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.

  Example:
  `comb default` of the experiment description file DFD_a from Example 6.5
  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 12.1 on page 203).
  - 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 12.3 hold.
  - When specifying **{<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 dfd.
  - When specifying **{<directory>/}<file_name> nonstrict** identical sample values for a factor are allowed. Experiment-specific multi-run operator dfd can not be applied in post-processing. Use UNC_MC operator ens instead.
  - During experiment post-processing restricted capabilities for the operator dfd apply for this experiment layout.

  Example: compare with the experiment description file DFD_d from Example 6.5
  Combination is implicitly as comb p1 , p2.
  Experiment description files DFD_b and DFD_d in Example 6.5
  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.

    Example: p2 * p1 is equivalent to p1 * p2

  - Factors that are scanned in parallel can be re-arranged in an arbitrary order.

    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.

  Example:

  If four factors are declared in the sequence p1 , p2 , p3 , p4
  then the explicitly stated <combination> p4 , p2 * p3 , p1
  is normalized to p1 , p3 * p2 , p4.

- Normalisation 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 $\{ x_{nj} \}_{j=1,\ldots,n^*}$ for $<x_n>$
  …
  loop over all factor sample values $\{ x_{2j} \}_{j=1,\ldots,2^*}$ for $<x_2>$
  loop over all factor sample values $\{ x_{1j} \}_{j=1,\ldots,1^*}$ for $<x_1>$
  endloop
  endloop
  …
  endloop
- For comb default :
  Is put down to comb <combination> (see above)

6.4.2 Example

Experiment description file DFD_a represents an experiment description according to Fig. 4.6 (a) on page 26, DFD_b and DFD_d according to Fig. 4.6 (b) and DFD_c according to Fig. 4.6 (c).

world.edf_DFD_a:

```
general          descr       Experiment description for the examples
general          descr       in the SimEnv User Guide (Fig. 4.6 (a))
general          type        DFD

factor     p1    descr       latitudinal phase shift
factor     p1    unit        pi/12.
factor     p1    type        add
factor     p1    default     1.
factor     p1    sample      list 1, 3, 7, 8              ... 2, 4, 8, 9 for p1

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
```

Results in adjusted factor values …

world.edf_DFD_b:

```
general          descr       Fig. 4.6 (b)
general          type        DFD

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
```

world.edf_DFD_c:

```
general          descr       Fig. 4.6 (c)
general          type        DFD

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
```

world.edf_DFD_d:

```
general          descr       Fig. 4.6 (b)
general          type        DFD              file world.dat DFD d:

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_DFD_d strict   ... (1,2), (3,3), (7,4),
                                                               (8,5) for (p1,p2)
```

Example files: world.edf_DFD_[a | b | c | d]

Example 6.5 *Experiment description files <model>.edf for DFD experiments*

6.5 Uncertainty Analysis – Monte Carlo Method UNC_MC

The experiment specific information for experiment description files in Tab. 6.1 on page 69 is defined for an UNC_MC experiment as describes in Tab. 6.8.

Tab. 6.8 *Experiment specific elements of an edf file for UNC_MC (line type: m = mandatory, o = optional)*

keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
factor	<factor_ name>	sample	m	1	[distr <distribution> | file {<directory>/} <file_name>]	distr: distribution and distribution parameters to derive a sample of values <factor_smp_val> file: file name to import an external sample of values <factor_smp_val>
		sample_ method	see ex-plan.	1	[pseudo | quasi | stratified]	sampling strategy (only for *sample* "distr <distribution>")
		sample_ include	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{n1}$>: <real_val$_{n2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to include for sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
		sample_ exclude	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{m1}$>: <real_val$_{m2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to exclude from sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
specific	<nil>	seed	o	1	[random | fixed]	method how to initialize the random number generator
		runs	m	1	<val_int>	number of runs to be performed for the experiment
		function	o	≥ 0	<result>	stopping function to use by the 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. Result 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. 6.8 the following additional rules and explanations apply:

- Sub-keyword ***runs***:
  Specify here the number of runs to be performed.
- For **all the other sub-keywords** check Section 6.1.1 on page 72.

6.5.1 Stopping Rule

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 performed 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 performed without applying the test in order to generate a stabilized stopping function sample res(z).

The whole experiment is stopped if

- the level of significance of the Pettitt test is below 0.05 for the already performed run ensemble and
- there were at least <val_int>/5 single runs after that single run that represents the point of change. <val_int> is the number of declared runs in <model>.edf (see above). This condition is introduced to avoid to run into a local point of change.

UNC_MC experiments with a stopping function cannot be restarted. 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

```
general       descr           Experiment description for the examples
general       descr           in the SimEnv User Guide
general       type            UNC_MC

factor   p1   descr           latitudinal phase shift
factor   p1   unit            pi/12.
factor   p1   type            add
factor   p1   default         1.
factor   p1   sample_method   stratified
factor   p1   sample          distr U(-6,6)  p1 is sampled from a uniform distribution
                                             between -6 and 6. In simenv_get_*
                                             each value is increased by 1.
factor   p2   type            multiply
factor   p2   default         2.
factor   p2   sample_method   pseudo
factor   p2   sample          distr N(2,.04) p1 is sampled from a normal distribution
                                             with mean = 1. and variance = 0.4.
                                             In simenv_get_* each sampled value is
                                             multiplied by 2.
factor   p2   sample_include  1.6:2.4        sample only within the 2σ interval
```

```
factor    p3   type            set
factor    p3   default         3.
factor    p3   sample          file world.dat_UNC_MC   sample for p3 is read from file
                                                        world.dat_UNC_MC
specific       runs            250
specific       function        avg(atmo_g)    apply avg(atmo_g) as stopping function
```

Example file: world.edf_UNC_MC

Example 6.6 *Experiment description file <model>.edf for UNC_MC*

6.6 Local Sensitivity Analysis LSA

The experiment specific information for experiment description files in Tab. 6.1 on page 69 is defined for a LSA experiment as follows:

Tab. 6.9 *Experiment specific elements of an edf file for LSA (line type: m = mandatory)*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
factor	<factor_ name>	sample		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 positive and ordered in a strictly monotonic increasing manner. (for syntax see Tab. 12.5)

To Tab. 6.9 the following additional rules and explanations apply:

- For a LSA experiment only the factor adjustment types “add” and “relative” are allowed.
- Values from the value list must be positive and unique.

6.6.1 Sensitivity Functions, Run Sequence

Example:

The absolute sensitivity function (cf Tab. 4.5 on page 30) 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
                        sample point
            endloop
endloop
loop        over negative increment sequence
            loop  over experiment factors
                        sample point
            endloop
endloop
```

6.6.2 Example

```
general          descr      Experiment description for the examples
general          descr      in the SimEnv User Guide
general          type       LSA

factor     p1    descr      latitudinal phase shift
factor     p1    unit       pi/12.
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_LSA

Example 6.7 *Experiment description file <model>.edf for LSA*

6.7 Bayesian Technique – Bayesian Calibration BAY_BC

Due to the sequential construction of the MCMC chain during the experiment a BAY_BC experiment can not be performed in parallel and can not be restarted.

Tab. 6.10 *Experiment specific elements of an edf file for BAY_BC (line type: m = mandatory, o = optional)*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
factor	<factor_ name>	sample	m	1	distr <distribution>	prior distribution to generate a representative sample of values <factor_smp_val> given the information defined in <model>.bdf
		sample_ include	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{n1}$>: <real_val$_{n2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to include for sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
		sample_ exclude	o	1	<real_val$_{11}$>: <real_val$_{12}$> { , … , <real_val$_{m1}$>: <real_val$_{m2}$> }	lower bounds <real_val$_{i1}$> and upper bounds <real_val$_{i2}$> of intervals to exclude from sampling according to distribution <distribution>. <real_val$_{i1}$> < <real_val$_{i2}$>
specific	<nil>	seed	o	1	[random | fixed]	method how to initialize the random number generator
		runs	m	1	<val_int>	chain length number of runs to be performed for the experiment will be smaller

To Tab. 6.10 the following additional rules and explanations apply:

- Sub-keyword ***sample***:
  Each factor in a BAY_BC experiment comes with its own (marginal) prior distribution which will be qualified in the course of experiment by generating a representative sample by taking into account measurement values and measurement errors from the real system.
- Sub-keyword ***runs***:
  Specify here the length of the chain of visited points in the factor space. The number of single simulation runs performed during the experiment is smaller than this length since simulation runs are not performed for candidate points that are outside the definition range of any individual factor according to its prior distribution and optional sample_include / sample_exclude intervals.
- Sub-keyword ***type*** (Section 6.1.1 on page 72):
  For BAY_BC only the factor adjustment type value “set” is allowed.
- For **all the other sub-keywords** check Section 6.1.1 on page 72.

6.7.1 Bayesian Calibration Description File <model>.bdf

Bayesian calibration is the only experiment type that comes with a mandatory SimEnv file which supplements the experiment definition file <model>.edf. This supplementary file is the Bayesian calibration description file <model>.bdf where the applied MCMC method, the covariance matrix for the normal multivariate jump and the likelihood function(s) are defined.

Tab. 6.11 *Elements of a BAY_BC bdf file*
(line type: m = mandatory, o = optional)

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
general	<nil>	descr	o	any	<string>	overall likelihood descriptions
		method	m	1	[metropolis_ standard | metropolis_ reflected]	Markov chain Monte Carlo MCMC method to use
		cov_matrix	o	1	file {<directory>/} <file_name>]	lower triangular matrix (including diagonal) of factor covariances
		jump_length	o	1	<val_float>	modifier for the length of the multivariate normal jump <val_float> > 0.
		settings	m	1	<val_int>	number of settings for a multiple setting BAY_BC experiment <val_int> ≥ 1
likelihood	<symbolic_ likelihood_ name>	descr	o	1	<string>	likelihood descriptions
		model_output	m	≥1	<result>	likelihood function model output part A result of any dimensionality formed according to the rules of the SimEnv post-processor. Result definition can be arranged at a series of function lines in analogy to the rules for result expressions (cf. Section 8.1.1).
		data	m	1	file {<directory>/} <file_name>]	likelihood function data part The name of an ASCII file that holds the measurement values (averages) and the corresponding errors (variances)

To Tab. 6.11 the following additional rules and explanations apply:

- Sub-keyword ***method***:
  The method metropolis_standard corresponds to the regular Metropolis-Hastings algorithm while for metropolis_reflected the so-called Metropolis-with-Reflection algorithm (check Section 4.7) is used. The latter method demands per factor only one resulting interval that is the intersection from the definition domain of the distribution and sample_include / sample_exclude intervals specified in <model>.edf.

- Sub-keyword ***cov_matrix***:
  Specify here the name of a file that holds the covariance matrix of all factors. The covariance matrix is used for the normal multivariate jump from an accepted point in the factor space to a candidate point. Only the lower triangular matrix including the diagonal with the factor variances has to be provided. Sequence of rows and columns of the matrix is determined by the sequence of factors in <model>.edf: Line number i (i=1,…,k for k factors) of the file has to have i values, where value no. j (j=1,…,i-1) is the covariance between factor i and factor j and the i-th value is the variance of factor i. For reflected Metropolis MCMC covariances have to be zero.
  In case the sub-keyword is not used factor covariances are set to zero and factor variances are derived from the corresponding distributions as defined by the sub-keyword *sample* in <model>.edf. *sample_include* / *sample_exclude* intervals as potentially also specified in <model>.edf are not taken into account for deriving the variances.
  For the file syntax check Section 12.3 on page 208.
- Sub-keyword ***jump_length***:
  From the last accepted point of the chain of visited points a new candidate point is constructed by a multivariate normal jump. The length of the jump can be modified by the specified multiplier value while the direction of the jump will not be changed. There are two major reasons for changing the jump length:
  (1) jump_length < 1: to reduce the number of candidates that are located outside the definition range of any individual factor and
  (2) jump_length > 1: to speed up the convergence of the experiment, i.e., explore the factor space adequately
  In case the jump_length is not specified a default value 1. is assumed, resulting in no modification of the jump length.
- Sub-keyword ***settings***:
  SimEnv allows for composing the likelihood function from performing a single model run for a number of different model settings where all settings consistently follow the single or multiple likelihood function case. This approach is called the multiple setting likelihood function case (check Section 4.7). Model settings in such a case may differ in model calibration or site conditions. While the result part of the likelihood function(s) is/are unique for all settings the data part(s) differ(s) among the settings. Settings always count from 1. Please check Section 6.7.2 below for more information how to set up such an experiment.
- Sub-keywords ***model_output*** and ***data***:
  For the Metropolis-Hastings algorithm applied in a BAY_BC experiment the posterior probability is the product of the prior probability and the likelihood function. The latter follows a normal distribution where the mean is the average measurement value and the variance is derived form the measurement error (check Section 4.7). The sub-keyword ***data*** supplies the data part of the likelihood function. Specify here the name of a file where the measurement data averages and variances are defined. With the sub-keyword ***model_output*** the model output part of the likelihood function is supplied. A model output variables and/or result of any dimensionality has to be specified here. Both parts are combined during performance of the experiment to compute successively an individual likelihood from a data pair with average and variance of the table and its corresponding element from the model output part. Correspondence is achieved by a sequential approach: The i-th individual likelihood is computed from the i-th data pair (the i-th record) and the i-th element of the result. Determination of the i-th element of multidimensional results follows the column-major order model (see Glossary in Section 15.7). Individual likelihoods are multiplied afterwards to form the final likelihood function and consequently the posterior probability as described in Section 4.7. The number of average-variance data pairs and the number of values of the result (= product of the extents) must be identical. For the structure of the data file check below.
  Besides this single likelihood function case (check Section 4.7) the user can specify in <model>.bdf for the multiple likelihood function case several likelihood functions with different <likelihood_name>. Generally, the likelihood functions differ in their data and result parts from each other and each likelihood function follows the rules for the single likelihood function case. Again, the single case likelihoods are multiplied to determine the multiple case likelihood.

The file for the data part of the likelihood function has to follow the syntax rules of ASCII data files in Section 12.3. Per data record one data pair has to be specified: The first value is the average of the measurement value and the second value is the assigned variance.
As the number of average-variance data pairs in the table and the number of values of the result must be identical nodata pairs have to be specified in the table for result elements where no data values are associated with. Use the float/real*4 nodata value as specified in <model>.cfg or the corresponding default nodata value (see Section 11.8). For data that are scarcely scattered compared to the number of result elements the shortcut

<val_int>*nodata

with <val_int> ≥ 1 can be used instead of specifying <val_int> succeeding nodata pairs

3.4E+38 3.4E+38
…
3.4E+38 3.4E+38

where 3.4E+38 is the default float/real*4 nodata value representation. If an other nodata value for float/real*4 is specified in <model>.cfg use that instead.

Keep in mind that the prior distribution is only determined by the marginal factor distributions as defined in <model>.edf. If for a factor sample_include or sample_exclude intervals are defined that alter the definition domain of the probability they are used to check whether a candidate point is outranged. In contrast, the prior distribution of the factor is not changed by defining such intervals for a factor distribution.

6.7.2 Multiple Setting Likelihood Function Case

The general idea behind this case is to derive the posterior distribution as a representative sample by taking into account several settings of a unique model rather than run the model only for a single setting. The latter is the standard case of SimEnv in terms of the model calls per SimEnv single run: for each single SimEnv run the model is performed once only. In contrast, for the multiple setting case each single SimEnv run is a series of the model runs of the individual settings. Model settings may differ in their calibration e.g., for site conditions if a model is considered to run at a local / regional scale. While the result part of the likelihood function(s) is/are unique for all settings the data part(s) differ(s) among the settings. In other words, setting-related likelihood functions are composed for each setting from unique result parts and setting-specific data parts. Overall likelihood function is the product of all individual setting-related likelihood functions.

The Bayesian calibration description file <model>.bdf for the multiple setting case differs only in the value of the sub-keyword *settings* from that for the other two cases. However, instead of checking and using the data files as specified by the sub-keyword *data* setting specific data files are checked and used. They are expected with names as declared in the *data* sub-keyword and appended by "**_s_<setting_number>**". Setting numbers count from 1.

For each setting a single model run is performed. To distinguish between the model outputs of the individual settings experiment output is stored for each setting to an individual subdirectory. Subdirectories are created in the model output directory as specified in <model>.cfg. Subdirectory names are "**s_<setting_number>**". As usual, experiment related output files are stored in the current workspace. Multiple setting experiments can not be post-processed in a common post-processing session. Instead, modify temporarily the model output directory in <model>.cfg to a setting subdirectory and post-process a single setting experiment output individually.

In the very artificial case the likelihood functions differ between settings, specify in <model>.bdf all likelihood functions and assign a table consisting only of nodata elements for those settings where an individual likelihood is not to be applied.

To interface the model to SimEnv for the multiple setting case the model wrap shell script <model>.run is used that represents a single model run. Since a single run for the multiple setting case is a sequence of individual single runs over all settings SimEnv provides a method how to deal with this feature:

Instead of directly calling the model within <model>.run the shell script

$SE_HOME/bin/simenv_bay_bc_sh <adapt_and_perform_setting_script>

is called.
The argument <adapt_and_perform_setting_script> to this script is the name of a shell script where the adaptation of the model to one individual setting and afterwards the performance of the adapted model for this individual setting have to be implemented. Within the shell script the operating system environment variable SE_BAY_BC_SETTING is defined. Its value is the number of the current setting. It has to be used for the adaptation.

As usual, the dot scripts $SE_HOME/bin/simenv_ini_sh and $SE_HOME/bin/simenv_end_sh have to be applied in <model>.run. Dot scripts $SE_HOME/bin/simenv_get_sh and $SE_HOME/bin/simenv_get_run_sh can be applied in <adapt_and_perform_setting_script>. If so, take care to perform $SE_HOME/bin/simenv_ini_sh before. $SE_HOME/bin/simenv_end_sh must not be applied in <adapt_and_perform_setting_script>.

6.7.3 Bayesian Calibration Log Files <model>.blog and <model>.bmlog

During the random walk by the Metropolis-Hastings algorithm a new candidate point in the factor space can be

- **outrange** as it is outside the definition domain of the distribution of any factor and will be discarded afterwards.
  For an outranged point a single simulation run will not be performed.
- **rejected** for a Metropolis ratio MR ≤ 1
  a candidate is rejected with a probability of 1-MR
  For a rejected candidate point a single run was performed.
- **accepted** always for MR > 1 or
  for MR ≤ 1 a candidate is accepted with a probability of MR
  For a accepted candidate point a single run was performed.
- **reflected** for an accepted candidate that was reflected before
  (only for the Metropolis_reflected method)
  For a reflected candidate point a single run was performed.

The log file **<model>.blog** records information how a candidate point was classified according to the above four categories. Additionally, the likelihood function value, the prior probability and the sampled values (and **not** the adjusted values by simenv_get_*) of the point are reported to this file. For outranged and rejected points the information from the previously accepted point is listed instead from the point itself. At the end of the file a statistical summary is attached. The sequence of the records in the file relates to the sequence of generated candidate points in the chain. The point with the default (nominal) factor values that correspond to the run number 0 is assumed as an accepted point.
The number of points records in <model>.blog is larger than the number of sample points in <model>.smp as for outranged candidates the model is not performed. <model>.smp is generated during performance of the experiment. <model>.blog is also exploited by the BAY_BC specific post-processing operators bay_bc_run_mask and bay_bc_run_weight (see Section 8.4.7).

For the multiple likelihood function case and the multiple setting likelihood function case another file **<model>.bmlog** is generated during experiment performance. It lists individual likelihood functions for each setting and/or each likelihood function. Likelihood function numbers correspond to the sequence of the likelihood functions as defined in <model>.bdf. Unlike <model>.blog this file lists also the candidate point likelihoods for rejected points. As the model is not run for outranged candidates the file can not supply information for such candidates.

6.7.4 Examples

```
general       descr          Experiment description for the examples
general       descr          in the SimEnv User Guide
general       type           BAY_BC

factor   p1   type           set
factor   p1   default        1.
factor   p1   sample         distr N(1,0.01)
factor   p1   sample_include 0.9:1.1        sample only within the 3σ interval

factor   p2   type           set
factor   p2   default        2.
factor   p2   sample         distr U(1.9,2.1)

factor   p3   type           set
factor   p3   default        3.
factor   p3   sample         distr N(3,0.01)

factor   p4   type           set
factor   p4   default        4.
factor   p4   sample         distr N(3.9,4.1)
factor   p4   sample_include 3.9:4.1        sample only within the 3σ interval

specific      runs           10000         chain length
```

Example file: world.edf_BAY_BC

Example 6.8 *Experiment description file <model>.edf for BAY_BC*

```
general                 descr          settings=1: multiple case
general                 descr                 =3: multiple setting case
general                 settings       1    or    3
general                 method         metropolis_reflected
general                 cov_matrix     file bay_bc.cov

likelihood   lf_atmo    model_output   sum_l('001',atmo(*,*,c=7,i=1:16))
                                                    16 values in total
likelihood   lf_atmo    data           world.dat_BAY_BC1

likelihood   lf_bios    model_output   sum_l('001',
likelihood   lf_bios    model_output   bios(c=10:-10,c=-10:10,i=1:6))
                                                    6 values in total
likelihood   lf_bios    data           world.dat_BAY_BC2
```

Example file: world.bdf

Example 6.9 *Bayesian calibration description file <model>.bdf*

<model>.run:

```
#! /bin/sh
. $SE_HOME/bin/simenv_ini_sh
# my_model is a model that is initialized (calibrated) for a setting
by the file setting_<setting_no>.ini

# only for the multiple case (running the model only in setting 1):
./my_model < setting_1.ini

# only for the multiple setting case (running the model for all settings):
$SE_HOME/bin/simenv_bay_bc_sh  my_model_setting_run.sh

. $SE_HOME/bin/simenv_end_sh
```

User shell script my_model_setting_run.sh
(for the multiple setting case: running my_model for one setting):

```
# $SE_BAY_BC_SETTING is the current setting number
# file setting_<setting_no>.ini holds setting-specific model information
./my_model < setting_$SE_BAY_BC_SETTING.ini
```

Excerpt from <model>.cfg:

```
model        out_directory            /scratch/modout
```

Experiment output is stored for the
- multiple case (and for other experiments with a single likelihood case) to /scratch/modout
- multiple setting case for setting 1 to /scratch/modout/s_1
- multiple setting case for setting 2 to /scratch/modout/s_2
- multiple setting case for setting 3 to /scratch/modout/s_3

Example 6.10 *<model>.run for BAY_BC*

Multiple case (settings = 1) data files for the likelihood function:

file world.dat_BAY_BC1
(16 data entries in total and
data pairs for time = 1 , 6 , 11 , 16:

```
111.1    0.111
4*nodata
112.2    0.112
4*nodata
113.3    0.113
4*nodata
114.4    0.114
```

file world.dat_BAY_BC2
(6 data entries in total and
data pairs for time = 1 , 3 , 6
float nodata value = 3.4E+38):

```
121.1    0.121
3.4E+38  3.4E+38
122.2    0.122
2*nodata
123.3    0.123
```

Multiple setting case (settings = 3) data files for the likelihood function:

for setting 1:
file world.dat_BAY_BC1_s_1 same as
file world.dat_BAY_BC1
(see <model>.run)

file world.dat_BAY_BC2_s_1 same as
file world.dat_BAY_BC2
(see <model>.run)

for setting 2:

file world.dat_BAY_BC1_s_2
(16 data entries in total and
no data pair:
likelihood function lf_atmo is not used for setting 2):

```
16*nodata
```

file world.dat_BAY_BC2_s_2
(6 data entries in total and
a data pair for time = 4)

```
3*nodata
221.1    0.221
2*nodata
```

for setting 3:

file world.dat_BAY_BC1_s_3
(16 data entries in total and
a data pair for time = 1):

```
311.1    0.311
15*nodata
```

file world.dat_BAY_BC2_s_3
(6 data entries in total and
a data pair for time = 6)

```
5*nodata
321.1    0.321
```

Example files: world.dat_BAY_BC1, world.dat_BAY_BC2

Example 6.11 *Data files for the likelihood functions of a BAY_BC experiment*
(For syntax of data file records check sub-keyword data in Section 6.7.1)

6.8 Optimization – Simulated Annealing OPT_SA

The experiment specific information for experiment description files in Tab. 6.1 on page 69 is defined for an OPT_SA experiment as follows:

To Tab. 6.12 the following additional rules and explanations apply:

- Sub-keyword ***seed***:
  This sub-keyword controls the seed of the used random number generator the simulated annealing algorithm. A random seed will result in different generated samples for repetitive performance of the same experiment while with a fixed seed the generated samples will be identical also across different machines (without taking into account numerical inconsistencies, e.g., rounding errors). If the sub-keyword is not stated initial seed is set to random.

Tab. 6.12 *Experiment specific elements of an edf file for OPT_SA (line type: m = mandatory, o = optional)*

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>	seed	o	1	[random | fixed]	method how to initialize the random number generator
		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. Result 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	<val_int>	number of single runs to end the experiment without checking the other optimization method related stopping criteria.

6.8.1 Special Features in OPT_SA

For an optimization experiment in general and for the coupled simulated annealing algorithm in particular the following features have to be kept in mind:

- This is an 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_sa_options.txt** from the sub-directory bin of the SimEnv home directory to <model>_opt_sa_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 of this file can be found in Ingber (2004).
- OPT_SA 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 OPT_SA experiment. The experiment always starts with run number 0 and ends if one of the criteria in the file [<model> | simenv]_opt_sa_options.txt (see above) is fulfilled or the explicitly stated end run number from the sub-keyword *runs* in <model>.edf is reached.
- As results of an OPT_SA 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 file <model>.fct.
- A protocol from the optimization procedure is made available by SimEnv in the ASCII file <model>.olog.

6.8.2 Example

```
general          descr      Experiment description for the examples
general          descr      in the SimEnv User Guide
general          type       OPT_SA

factor     p1    descr      latitudinal phase shift
factor     p1    unit       pi/12.
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       500
```

Example file: world.edf_OPT_SA

Example 6.12 *Experiment description file <model>.edf for OPT_SA*

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 node in foreground or in background and controlled by a Distributed Resource Manager (synonym: job system) SimEnv supports the Distributed Resource Managers load leveler LoadL and PBS/Torque. Experiment performance on the login node is organized in a way that the single runs of the experiment are performed sequentially. Experiments under a Distributed Resource Manager control enable assignment of the simulation load of the single runs of the experiment to a number of processor cores in distributed, parallel or sequential mode.

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

- For all experiment settings the user model has to be wrapped in a shell script <model>.run (cf. Section 7.2 and also Fig. 5.1).
- The model variables to be output during experiment performance are declared in the model output description file <model>.mdf. Output of such declared model variables 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.
- 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.
- Model output from run number <simenv_run_int> is stored in the file <model>.out[all | <simenv_run_char>].[nc | ieee]. For all experiment types a run number 0 with the default (nominal) values of all experiment factors is declared additionally to the runs described 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.
- The status of any running experiment can be acquired by the SimEnv service simenv.sts. For more information check Tab. 11.4. After the experiment has been finished an email is sent on demand (cf. Section 11.1) to the address as specified in <model>.cfg.
- 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 restarted for successive performance of experiment slices and/or after abnormal 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 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 59 and Example 7.1 below).
- Terminal output from <model>.run is redirected to the model native output log file <model>.nlog.
- 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. Use the corresponding SimEnv function simenv_get_run_* to get the current run number <simenv_run_char>. Cf. Tab. 5.8 on page 59 and Example 7.1 below.
  From the cancelled experiment only those single runs are available for experiment post-processing that were finished before the cancelled single run. Check the experiment log file <model>.elog to identify these single runs.
- SimEnv supplies a shell script simenv_kill_process to kill processes of models / programs that were started within <model>.run and that consumed more than a given threshold of CPU time. For example, with this script the process associated with the model run that does not converge and would run infinitely can be killed.
  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 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.
- For GAMS models and for the experiment type BAY_BC with the multiple setting case <model>.run has to have a special pre-defined structure. For GAMS models check Section 5.7.1 and for the experiment type BAY_BC with the multiple setting case check Section 6.7.2 for more information.

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). After the experiment has been finished the experiment can be wrapped up with the optional model / experiment specific shell script **<model>.end**.

- 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.
- 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.

The shell scripts <model>.run, <model>.ini, and <model>.end 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, Parallelization

According to the general SimEnv approach how to design an experiment the single runs of an experiment are independent from each other. The only exceptions are the experiment types OPT_SA and BAY_BC 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 node
  in foreground, background and distributed mode and
- distributed on a compute cluster under control of a Distributed Resource Manager
  in parallel, distributed and sequential mode

For an experiment performance controlled by a Distributed Resource Manager or on the login node in background **make sure that the environment variable SE_HOME is set correctly in the file $HOME/.profile**.

7.3.1 Local Experiment Performance on the Login Node

Two different distribution strategies are offered by the simulation environment for running an experiment on the login node:
Perform the single runs of an experiment …

- … sequentially on the login node
  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 node the experiment is started from.
- … distributed on the multicore login node
  mcore - multi-core sub-mode:
  For a multi-core processor login node 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 node the experiment is started from. As for background sub-mode, in multi-core sub-mode the login session can be closed after experiment preparation.

7.3.2 Experiment Performance controlled by the Distributed Resource Manager

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 of an experiment for compute cluster architectures. Currently, IBM's Job management system load leveler LoadL with the parallel operating environment POE and the Linux tools PBS/Torque are supported as Distributed Resource Managers DRM. For DRMs the processors of a compute cluster are assigned to job classes / queues where jobs can be submitted to.

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

- … on all the available processor cores of a job class / queue
  dis - distributed sub-mode:
  The single runs are submitted to the job class / queue 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 / queue 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 / queue is free.
  Use this sub-mode for best utilization of all job class processor cores.

- … on pre-allocated processor cores of a class / queue
  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 / queue. During the experiment one communication processor core 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 available in this class / queue. 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:
  SimEnv also offers a sequential sub-mode under control of the Distributed Resource Manager: One processor core of a job class / queue is assigned to the whole experiment and the experiment is performed sequentially on this processor core. The experiment performance will start when one processor core of this job class / queue is available.

After an experiment was submitted to the Distributed Resource Manager the current login session can be closed.

Default job control files are supplied by SimEnv to ensure communication with the Distributed Resource Manager. 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 the SimEnv parallelization strategies.
If necessary, copy the ASCII job control file **simenv_[dis | par | seq]_[aix | linux].[jcf | pbs]** from the sub-directory bin of the SimEnv home directory to <model>_[dis | par | seq]_[aix | linux].[jcf | pbs] in the current workspace, modify the file according to the needs of the model one wants to perform and / or the node one wants to use and start afterwards simenv.run and/or simenv.rst. If available in the current workspace, the modified job control file is used instead of the original file in the sub-directory bin of the SimEnv home directory. simenv_[dis | par | seq]_[aix | linux].[jcf | pbs] and/or <model>_[dis | par | seq]_[aix | linux].[jcf | pbs] submit a job in distributed / parallel / sequential sub-mode under Distributed Resource Manager control.

The default job control files do not enable automatic restart of the experiment by the Distributed Resource Manager after an abnormal end. The user has to restart the experiment manually after such an event.

For performing a parallel model itself in a DRM environment see Section 5.11.

7.3.3 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 Section 11.1).

A partial experiment performance is also possible for an experiment restart. Experiment partial performance is not possible for the experiment types UNC_MC with a stopping function, BAY_BC and OPT_SA.

For more information check Fig. 7.1.

7.3.4 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 node 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 node and/or on a compute cluster
  The above solution fails 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.

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*

7.3.5 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 compute 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:

```
login02:~> 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 authorised_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.

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 7.3.3) it can be restarted. Restart can be applied to an experiment several times successively.

- Simply restart the experiment by simenv.rst. The SimEnv service simenv.rst has the same usage as simenv.run. Do not change 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. Experiment restart works without standard SimEnv experiment preparation. Instead, experiment preparation files and other information from the interrupted experiment will be used. The configuration file **<model>.cfg** will be checked anew for experiment restart.
- Dependent on the experiment log file <model>.elog, written by the previous / interrupted experiment a single model run out of run ensemble as defined in <model>.cfg for the experiment to restart will be
  - Performed if this run has neither a start nor a finish entry in the elog file
  - Not performed if this run has a start and a finish entry in the elog file
  - Performed anew if the run has a start entry but no finish entry in the elog file.
- For the latter case, a model restart shell script **<model>.rst** can be optionally provided by the user to prepare the 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 59 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. 11.10).
  Terminal output from <model>.rst is re-directed to the log file <model>.nlog.
- For a restart, the optional user-defined experiment preparation shell script **<model>.ini** will be performed only on demand. The corresponding 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)

- Restart can be launched in a different mode (on the login node or under job Distributed Resource Manager control, the latter also in an other job class / queue) and sub-mode and/or on an other node than that of the previous / interrupted experiment.
- Minutes of the restarted experiment will be appended to the log files <model>.mlog, <model>.nlog, and <model>.elog, respectively from the previous / interrupted experiment.
- Experiment restart is not possible for the experiment types UNC_MC with stopping rule, BAY_BC and OPT_SA.

For the model world_sh (cf. Example 15.10 on page 236) 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 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 restart <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 | pbs]	model-specific job control file to submit an experiment in distributed, parallel and/or sequential sub-mode under Distributed Resource Manager control (jcf for LoadL, pbs for PBS/Torque) • Copy from $SE_HOME/bin/simenv_[dis | par | seq]_ [aix | linux].[jcf | pbs] if required	D	optional
<model>_opt_ options.txt	model-specific control and option file for experiment type OPT_SA • Copy from $SE_HOME/bin/simenv_opt_sa_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 node or under Distributed Resource Manager control*
D: *experiment performance under **D**istributed Resource Manager control*
O: ***O**PT_SA experiment performance*

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

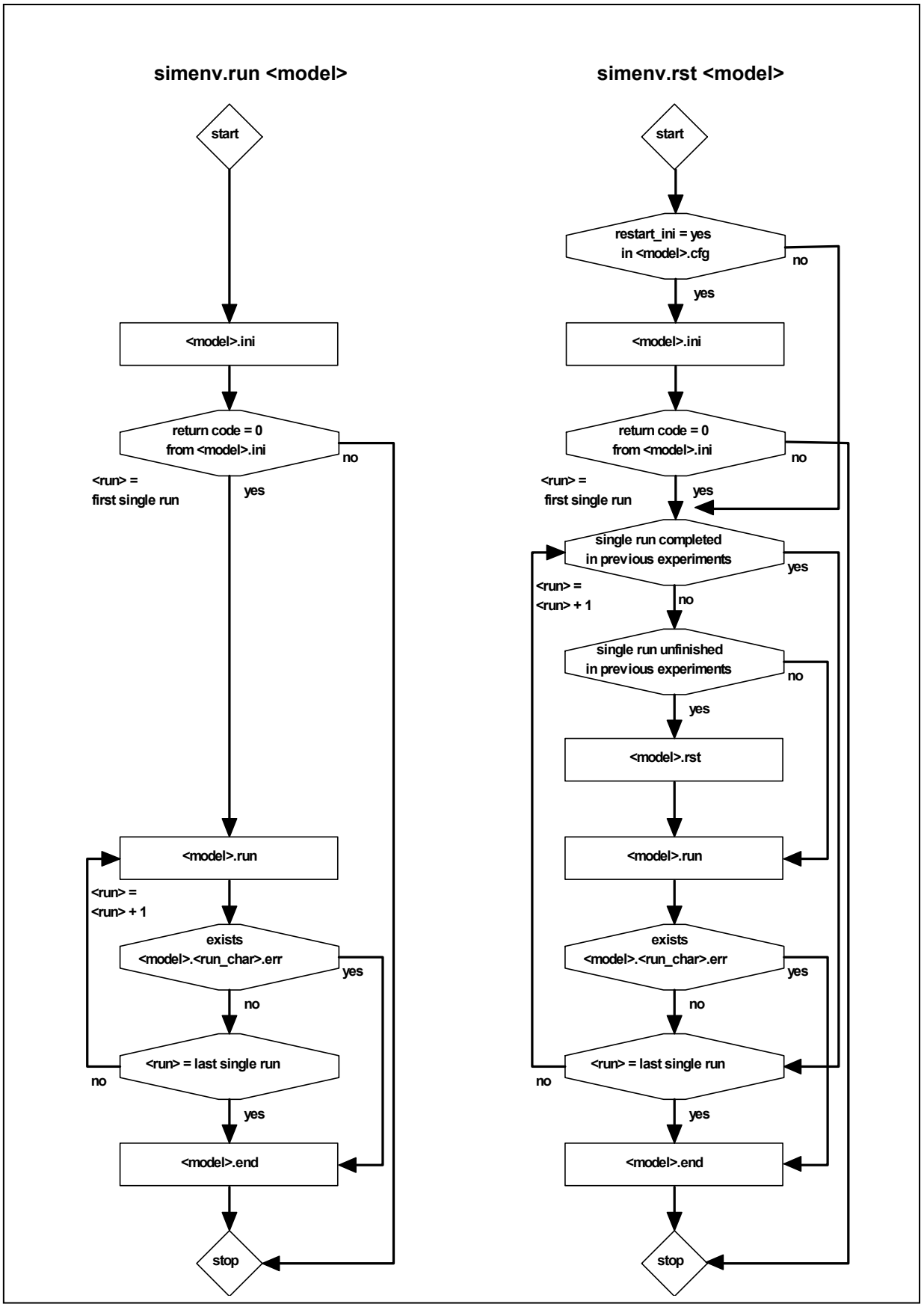


Fig. 7.1 *Flowcharts for performing simenv.run and simenv.rst*
First and last single run always refer to the corresponding settings in <model>.cfg

7.6 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 UNC_MC with stopping rule, BAY_BC, and OPT_SA
<model>.blog	for BAY_BC
Optional from the current workspace	
<model>.elog	
<model>.mlog	
<model>.nlog	
<model>_[dis | par | seq]_[aix | linux].[jcf | pbs]	
<model>.bmlog	for BAY_BC
<model>.olog	for OPT_SA
<model>.opt_sa_options.txt	for OPT_SA

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 experiment 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 experiment output of the experiment and from data. Generally speaking, a post-processor result is specified by a chain of operators which are applied to model output variables and/or data. This operational description of the result is optionally prefixed by a result descriptor where a name and/or attributes can be assigned to the result.

<result> := { <result_descriptor> := } <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> instead of a result, possibly prefixed by white spaces.
For case sensitivity of <result> check Tab. 11.12 on page 197.

<result_descriptor> = ({ ‘result_name’ } , { ’result_description’ } , { ’result_unit’ })

Optionally describes the name of the result and its description and unit in the corresponding result output file (cf. Chapter 10). For one post-processor output file (one post-processor session) ‘result_name’ has to be unique.
The result descriptor with its separator “:=” has to be specified in the first input line. The result expression itself may follow at the following input line. ‘result_description’ and ‘result_unit’ are case sensitive, ‘result_name’ is transformed to lower cases. All three properties can be empty and/or a character string with white spaces).
In the absence of ‘result_description’ and ‘result_unit’ there are default values for them if the result expression is a special type of a single operand – single operator expression. In the absence of ‘result_name’ there is a general default name. All three strings can comprise wildcard substrings. For all defaults and wildcard see below.

<result_expression> is a chain of SimEnv and user-defined operators applied to model output variables and/or 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.

For a single operand – single operator result expression and in absence of ‘result_description’ and/or ‘result_unit’ in the result descriptor both properties may be substituted by defaults.
Here, a single operand – single operator result expression is defined as a result without any operator or only from one operator and using exactly one model output variable and/or one experiment factor. For such operators that are invariant with respect to the unit of the operand ‘result_description’ and/or ‘result_unit’ are 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). For all other cases <result_description> and <result_unit> are blank strings / undefined. For a list of invariant operators check Section 15.5.5.

The default value for ‘result_name’ is ‘res_<digit><digit>’ where <digit><digit> where the first result in an post-processing output file with a default name is ‘res_01’ and <digit><digit> can count up to 99.
If the result expression is composed from wildcard operands &v& and/or &f& (see Sections 8.1.2 and 8.7) then the three elements of the optional result descriptor can also comprise these wildcard.

Having a model output variable definition as in Example 5.1 on page 44 then in experiment post-processing

```
abs(atmo)+3                    (multi-operand – multi-operator result expression)
                               applies operator abs to atmo and adds 3
                               <result_name> = ‘res_01’
                               <result-description> (undefined)
                               <result_unit> (undefined)
('Energy',' ','MWh') := abs(atmo)+3    as above, but:
                               <result_name> = ‘energy’
                               <result_unit> = ‘MWh’
('Energy',' ','MWh') :=        as above
abs(atmo)+
3
atmo                           (single operand – single operator expression)
                               Applies no operator to variable atmo
                               <result_name> = ‘res_02’
                               <result_description> = ‘aggregated atmospheric state’
                                        (according to <model>.mdf)
                               <result_unit> = ‘atmo_unit’
                                        (according to <model>.mdf)
abs(atmo)                      (single operand – single operator expression)
                               Applies operator abs to atmo to variable atmo
                               Operator abs is invariant w.r.t. the unit of its operand
                               <result_name> = ‘res_03’
                               <result_description> = ‘aggregated atmospheric state’
                                        (according to <model>.mdf)
                               <result_unit> = ‘atmo_unit’
                                        (according to <model>.mdf)
sign(atmo)                     (single operand – single operator expression)
                               Applies operator sign to variable atmo
                               Operator sign is not invariant w.r.t. the unit of its operand)
                               <result_name> = ‘res_04’
                               <result_description> = (undefined)
                               <result_unit> = undefined
('Energy','new description',' ' := abs(atmo)
                               <result_name> = ‘energy’
                               <result_description> = ‘new description’
                               <result_unit> = ‘atmo_unit’
                                        (according to <model>.mdf)
```

```
('Energy','new description',' ' := sign(atmo)
                                  <result_name> = 'energy'
                                  <result_description> = 'new description'
                                  <result_unit> (undefined)
```

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`

- Integer constants <val_int> or real constants <val_float>
  In the following abbreviated by **int_arg and/or real_arg**

  Example: `12` and `-12` or `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 8.6)

  Example: `equ_100yrs_m`

- Wildcard operands — (cf. Section 8.7)

  Example: `&v&`

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

- A dimensionality **dim > 0**
  Extents **ext(i) > 1** with i=1 ,..., dim
  Coordinates **coord(i)** with i=1 ,..., dim

  are assigned to (cf. Section 5.1). 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. 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 extents and 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 42) by their

- Name
- Data type (cf. Tab. 5.4 on page 43).
- Dimensionality
- Extents
- Coordinate assignment to each dimension
- Index assignments to each dimension
- Use the service simenv.chk to check variables description in model output description file <model>.mdf

Addressing model output data fields or parts of it in experiment post-processing is done 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 = * denotes the complete range of a dimension.
- 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 = * denotes the complete range of a dimension.
- Dimensions with their individual index / 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.
  Or each dimension c= * is identical to i= * is identical to *
- In the general SimEnv configuration file <model>.cfg (cf. Section 11.1 on page 181) 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 44 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)
                                  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. 11.11) or form new coordinates from resulting factor adjustments. 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.

  Example: `ens(atmo)`

SimEnv experiment post-processing operators may have two special types of arguments / operands:

- 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., dfd)

  Example:
  ```
  count('undef',atmo)
  dfd(' ',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 arg 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 for built-in or user-defined operators the standard approach for description of variables. 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: for the operator addition with two operands atmo_g and 345

```
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.

The requirement for a lot of operators to have same coordinates for same dimensions may strongly restrict application of experiment post-processing especially for hypothesis checking. 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.

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

Argument restriction(s) / result description type number		**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)		dimensionality, extents and coordinates of the only non-character / non-constant argument arg can be arbitrary	dimensionality is higher than that of the only non-character / non-constant argument. Extents and coordinates of this argument are copied to the result, appended dimensions are explained individually. dim_{res} > dim_{arg} $ext_{res}(j)$ = $ext_{arg}(j)$ for all j $coord_{res}(j)$ = $coord_{arg}(j)$ for all j
(7)		only character arguments, constant arguments or without arguments	dim_{res} = 0

Having a model output variable definition as in Example 5.1 on page 44 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 operator dfd of experiment type DFD:

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
count	number of all, all defined or all undefined values
maxprop	maximal, suffix related property of values
minprop	minimal, suffix related property of values
hgr	histogram (heuristic probability density function) of values

For more information check Sections 8.3.3.

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 118)	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
+ arg1	identity (monadic +)	(1)		fourth
– arg1	negation (monadic -)	(1)		fourth

The following explanations hold for the operators in Tab. 8.3:

- 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 118)	Argument value restriction	Example
Basic operators				
abs(arg1)	absolute value	(1)		`abs(-3) = 3.`
dim(arg1,arg2)	positive difference	(2)		`dim(10,5) = 5.` `dim(5,10) = 0.`
exp(arg1)	exponential function	(1)		`exp(1.) = 2.7183`
int(arg1)	integer truncation value	(1)		`int(7.6) = 7.` `int(-7.6) = -7`
log(arg1)	natural logarithm	(1)	arg > 0	`log(2.7183) = 1.`
log10(arg1)	decade logarithm	(1)	arg > 0	`log10(10) = 1.`
mod(arg1,arg2)	remainder	(2)	arg2 ≠ 0	`mod(10,4) = 2.`
nint(arg1)	nearest integer value	(1)		`nint(7.6) = 8.` `nint(-7.6) = -8.`
round(int_arg1, arg2)	round arg2 to int_arg1 decimal places	(1)	1 ≤ int_arg ≤ 6	`round(1,2.34) = 2.3` `round(1,2.36) = 2.4`

sign(arg1)	sign of value	(1)		`sign(-3) = -1.`
				`sign(0) = 0.`
				`sign(3) = 1.`

sqrt(arg1)	square root	(1)	arg ≥ 0	`sqrt(4) = 2.`
Trigonometric operators				
sin(arg1)	sine	(1)		`sin(0) = 0.`
cos(arg1)	cosine	(1)		`cos(0) = 1.`
tan(arg1)	tangent	(1)	arg ≠ π/2±n*π	`tan(0) = 0.`
cot(arg1)	cotangent	(1)	arg ≠ ±n*π	`cot(1.5708) = 0.`
asin(arg1)	arc sine	(1)	abs(arg1) ≤ 1	`asin(0) = 0.`
acos(arg1)	arc cosine	(1)	abs(arg1) ≤ 1	`acos(1) = 0.`
atan(arg1)	arc tangent	(1)		`atan(0) = 0.`
acot(arg1)	arc cotangent	(1)		`acot(0) = 1.5708`
sinh(arg1)	hyperbolic sine	(1)		`sinh(0) = 0.`
cosh(arg1)	hyperbolic cosine	(1)		`cosh(0) = 1.`
tanh(arg1)	hyperbolic tangent	(1)		`tanh(0) = 0.`
coth(arg1)	hyperbolic cotangent	(1)	arg1 ≠ 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.

  Example: `sin(bios(*,*,*))` = "sin(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.

8.3.3 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
  The result is a scalar (an operator result of dimensionality 0)
- Appending **suffix _n** (_n stands for n arguments):
  Aggregate an arbitrary number of non-character / non-constant arguments element by element.
  Currently, only operators min_n, max_n, minprop_n, and maxprop_n are implemented.
  The result has same dimensionality, extents and coordinates as the arguments.
- Appending **suffix _l** (_l stands 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. These operators correspond to the group-by clause of the standard query language SQL of relational database management systems.
  The result has a lower dimensionality as the only non-character argument according to the loop character argument.
- Appending **suffix _e** (_e stands for run ensemble):
  Aggregate separately for each element of the only non-character / non-constant argument over the ensemble of all single simulation runs but single run 0. Such operators are multi-run operators (check Section 8.4.1).
  The result has same dimensionality, extents and coordinates as the argument.
- Exceptions to the above rules hold for operators minprop, maxprop and hgr.
- Additionally, the operators can be used in the selection / aggregation filter argument char_arg1 of the multi-run **operator dfd** for the experiment type DFD. For details check Section 8.4.3.
- For **examples** without suffix and with suffix _n, _l check Section 8.3.5, with suffix _e check Section 8.4.5.

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

Aggregation and moment operator	**Argument restriction(s) / result description (Tab. 8.1, page 118)**
max(arg1)	(3)
min(arg1)	
sum(arg1)	
avg(arg1)	
var(arg1)	
avgg(arg1)	
avgh(arg1)	
avgw(arg1, arg2)	(4.1) arg2 = weight
count(char_arg1, arg2)	(3) char_arg1 = ['all' | 'def' | 'undef']
maxprop(arg1)	for dim_{arg1} > 1 dim_{res} = 1 $ext_{res}(1)$ = dim_{arg1} else dim_{res} = 0 return the index of that element of arg1 where the extreme is reached the first time according to the processing sequence of the argument field arg by the column-major order model (cf. Section 15.7 – Glossary).
minprop(arg1)	
hgr(char_arg1, int_arg2, real_arg3, real_arg4, arg5) one heuristic pdf from all values of arg5	dim_{res} = 1 $ext_{res}(1)$ = number of bins (no._of_bins) $coord_{res}(1)$ = char_arg1 for char_arg1 = 'bin_no' (bin number): $coord_{res}(1)$ values = equidist_end 1(1) no._of_bins for char_arg1 = 'bin_mid' (bin mid): $coord_{res}(1)$ values = equidist_end 1st_bin_mid (bin_width) no._of_bins char_arg1 ['bin_no' | 'bin_mid'] int_arg2 = no._of_bins: 4 ≤ int_arg2 ≤ no._of_values or = 0: automatic determination: no._of_bins = max(4,no._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

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 118)
max_n(arg1 , arg2 {, arg3 ,..., argn})	(4)
min_n(arg1 , arg2 {, arg3 ,..., argn})	
maxprop_n(arg1 , arg2 {, arg3 ,..., 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 , arg2 {, arg3 ,..., 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
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 $ext_{res}(dim_{res}) = dim_{arg2}$ $coord_{res}(dim_{res})$: name = index values =equidist_end 1(1)"n"	return the indices of those elements of arg2 where the extreme is reached the first time according to char_arg1 and to a For-tran-like processing sequence / storage model (cf. Section 15.7 – Glossary) of the argument field arg2.
maxprop_l(char_arg1, arg2)		

Aggregation and moment operator	Argument restriction(s) / result description
hgr_l(char_arg1, char_arg2, int_arg3, real_arg4, real_arg5, arg6) heuristic pdfs separately calculated for each index of the those dimensions of arg5 that remain according to char_arg1.	dim_{res} = 1 + dim_{res} of all other operators of this Tab. $ext_{res}(dim_{res})$ = number of bins no._of_bins $coord_{res}(dim_{res})$ = char_arg2 for char_arg2 = 'bin_no' (bin number): $coord_{res}(dim_{res})$ values = equidist_end 1(1) no._of_bins for char_arg2 = 'bin_mid' (bin mid): $coord_{res}(dim_{res})$ values = equidist_end 1st_bin_mid (bin_width) no._of_bins char_arg2 ['bin_no' | 'bin_mid'] int_arg3 number of bins 4 ≤ int_arg3 ≤ no._of_values_of_arg6 or 0: automatic determination = max(4,no._of_values_of_arg6/10) 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

The loop character argument char_arg1 is characterised as follows:

- The length of the string is equal to the dimensionality of the non-character / non-constant 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 formed only of 0 ('0…0') or 1 ('1…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 118)	Argument value restriction	Example

classify(
int_arg1,
real_arg2,
real_arg3,
arg4)
	classify arg4 into int_arg1 classes;
	potentially restrict classification 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 determination: number of classes = max(2,nmb of values/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 (compute partial sums of) arg2 according to char_arg1	(1)
		
		dim_{arg2} > 0
		char_arg1 = cumulation indicator per dimension
				`cumul(`
				`'0001',`
				`atmo)`

distr_par(char_arg1, int_arg2)	get for experiment types with probabilistic sampling the value of a factor's distribution parameter	(7) char_arg1 = factor name int_arg2 = number of the distribution parameter ≤ 2		`distr_par(` `'p1',2)`

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)`

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(`
				`'netcdf',`
				`'data.nc',`
				`'data.def',`
				`variable)`

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 118)	Argument value restriction	Example

get_experiment(
char_arg1,
char_arg2,
char_arg3,
arg4)
	include an other post-processed experiment output	(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)`

{un}mask_file(
char_arg1,
arg2)
	mask values of arg2 (set them undefined) as specified in the file char_arg1	(1)
		
		char_arg1 = mask file name
				`mask(`
				`'mask.dat',`
				`atmo)`

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 118)	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 ≤ no._of_runs
		addit. for DFD and LSA:
		= <filter argument>
		same as filter argument of operators dfd and
		['sens' | 'lin' | 'sym']'_'
		['abs' | 'rel']
		(cf. Sections 8.4.3 and 8.4.5)
				`run(`
				`'0',`
				`atmo)`
				`run(`
				`'sel_t(p1(4))',`
				`atmo)`

run_info(char_arg1)	number of the current single run and/or total number of runs of the experiment	(7) 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	(7)		`undef()`
usage(char_arg1)	get usage of a post-processing operator	(7) char_arg1 = operator name or ['all' | 'basic' | 'trigonometric' | 'aggr/moment' | 'advanced' | 'multi-run' | 'user-defined'] for the corresponding operator classes		`usage('run_info')` `usage('advanced')`

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.

- **Operator classify**
  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 equal zero then min(arg4) forms the lower bound of class number 1 and max(arg4) forms the upper bound 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 bound range.

- **Operator clip**
  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 114). 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.

- **Operator cumul**
  The operator cumul cumulates (computes partial sums of) 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.

- **Operator distr_par**
  The operator distr_par is applicable for all experiment types that demand explicitly a probabilistic sampling scheme (experiment types GSA_EE, GSA_VB, UNC_MC, and BAY_BC). The operator provides the value of a distribution parameter of a factor as specified in the experiment description file <model>.edf. The operator is not applicable to factors where the sample is imported from a file.

- **Operator flip**
  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.

- **Operator get_data**
  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. 12.3. Currently, ASCII and NetCDF files are supported (char_arg1 = ['ascii' | 'netcdf']).
  For ASCII data files the file syntax rules from Section 12.3 are valid. Since the ASCII data file itself does not come with any structure and coordinate information a coordinate specification / transformation file can be specified by the character argument char_arg3. For ASCII data files this argument is a mandatory one. It follows the same rules as for any coordinate transformation file (cf. Section 12.2). Keywords *general*, *assign*, and *coordinate* and the appropriate sub-keywords from Tab. 12.4 can be used to struc-

ture 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 column-major order 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 44 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*

- **Operator get_experiment**
  The operator get_experiment is to access to external SimEnv experiment 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 re-

sult 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 12.2. For restrictions in the path to the directory of the character arguments char_arg1 and char_arg3 check Tab. 12.3.

Attention:

Make sure

- that 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.

- **Operator get_table_fct**
  With the operator get_table_fct a table function 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 name of the table function file. It must be an ASCII file with two columns: The first column 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 12.3). For restrictions in the path to the directory of the character argument char_arg1 see Tab. 12.3. Check the table function world.dat_adv in the example directory $SE_HOME/exa of SimEnv for more information.

- **Operator if**
  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')

- **Operator mask**
  The operator mask supplies a method to mask (to set undefined) elements based on their 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 = '!=')

- **Operators mask_file** and **unmask_file**
  The operators mask_file and unmask_file mask the elements of arg2 (set them undefined) according to their position, the latter stored in the ASCII file char_arg1. For operator mask_file all those values of arg2 are masked that are specified in the mask file char_arg1 while for operator unmask_file all values of arg2 are masked that are not specified in the file char_arg1. Each line of the mask file char_arg1 has to follow the subsequent syntax:

  { [i | c] = } <value$_{11}$> { : <value$_{12}$> } <sep> … <sep> { [i | c] = } <value$_{n1}$> { : <value$_{n2}$> }

  It uses the syntax for model output variables as described in Section 8.1.3 on page 114. In particular, i= stands for local index addressing, c= for local coordinate addressing and if they are missing the global default as defined in the general SimEnv configuration file <model>.cfg (cf. Section 11.1 on page 181) for index and/or coordinate addressing is used. Additionally, a local (file) default can be set by lines

  i=

  or

  c=

  indicating that all addresses in the following records without a local setting are interpreted in the appropriate manner.
  n corresponds with the dimensionality of arg2, <value$_{i2}$> = <value$_{i1}$> if <value$_{i2}$> is missing and * instead of <value$_{i1}$> { : <value$_{i2}$> } is allowed (i=1,…,n). Comment lines starting with the first non-white space character # are allowed as for each ASCII data file (cf. Section 12.3 on page 208).
  Each line defines for the argument arg2 a block of values that are masked (for operator mask_file) or not masked (for operator unmask_file).
  The line

  <sep> … <sep> *

  in the mask file char_arg1 results for operator mask_file that all values of arg2 are masked and for the operator unmask_file in the identity of argument arg2.
  For restrictions in the path to the directory of the character argument char_arg1 see Tab. 12.3.

  Having a model output variable definition as in Example 5.1 on page 44 and assuming

  a data file **mask.dat** as

```
# this file masks the variable atmo
# tropical Africa and Arabia, level 2, last two decades:
c=
20:-20  -20:60  7  19:20
# this is equivalent to:
i=
17:27  41:50  2  19:20
# this is equivalent to:
c=20:-20  c=-20:60  2  19:20
#
# Antarctica, all levels and decades:
c=
-60:-88  *  *  *
# this is equivalent to:
i=
38:45  *  *  *
# this is equivalent to:
c=-60:-88  1:90  1:4  1:20
```

  then

```
mask_file('mask.dat',atmo)
```

  masks for atmo the above two regions with the declared levels and decades and

```
unmask_file('mask.dat',atmo)
```

  masks for atmo all but the two regions with the declared levels and decades.

Example 8.5 *Experiment post-processing operators {un}mask_file*

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

- **Operator move_avg**
  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 successive manner.

> 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) .

Example 8.6 *Operator move_avg*

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.

- **Operator rank**
  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 $arg2_1$ and $arg2_2$ of arg2 that are identical or have a maximum relative difference of $abs(arg2_1-arg2_2) / arg2_1 < 10^{-6}$:

```
Assume an argument arg2 with 6 values              ( 4., 2., 4., 4., 4., 8.)

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
```

Example 8.7 *Operator rank*

- **Operator regrid**
  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 12.2. For restrictions in the path to the directory of the character arguments char_arg1 check Tab. 12.3.

- **Operator run**
  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 DFD and LSA a run number unequal 0 can be selected implicitly by applying a filter of the corresponding operators (cf. Sections 8.4.3 and 8.4.5) as char_arg1 of the operator run.
  The file <model>.smp holds the sampled factor values to be adjusted by the default (nominal) 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 69. For examples see Example 8.12 and Example 8.13.

- **Operator run_info**
  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.

- **Operator transpose**
  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.

- **Operator undef**
  The operator undef supplies a 0-dimensional result as undefined. Among others, this operator can be used as an argument for the if-operator.

- **Operator usage**
  The operator usage prints a short usage for the post-processing operator name char_arg1. Additionally, operator class names can be used instead of an operator name:

'all'	for all	operators
'basic'	for basic	operators (Section 8.3.2)
'trigonometric'	for trigonometric	operators (Section 8.3.2)
'aggr/moment'	aggregation/moment	operators (Section 8.3.3)
'advanced'	advanced	operators (Section 8.3.4)
'multi-run'	multi-run	operators (Section 8.4)
'user-defined'	user-defined	operators (Section 8.5)

  The operator is a stand-alone operator: It must not be operands of any other operator.

- For (additional) **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 44 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_adv',atmo)
```
Operator table_fct with table world.dat_adv 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 8.8 *Experiment post-processing with advanced operators*

8.4 Built-In Experiment Specific Operators

8.4.1 Multi-Run Operators

Experiment specific operators are to navigate and process in the factor space according to the layout of the individual experiment type. A multi-run operator operates on one or more operands for all runs of the whole run ensemble or parts of it. The multi-run operator gets its result from the all single runs of the experiment that have to be taken into account. By contrast, a non multi-run (single run) operator gets its results individually from each single run and assigns this result also to that single run.

Addressing an operand within a multi-run operator normally results in application of the operator on the operand values from the whole run ensemble or parts of it and in aggregating across the run ensemble according to the operator. Addressing an operand outside a multi-run operator results in using the operand for the default run number 0 of the experiment.

The majority of the built-in experiment specific operators are multi-run operators. Exceptions are referenced explicitly.

Experiment specific operators for UNC_MC can be applied for all other experiment types. Keep in mind that for the other experiment types generally the factors do not follow a pre-defined distribution.

Multi-run operators can not be applied recursively.

For experiment type UNC_MC one of the multi-run operators is

avg_e(arg) that computes for each element the run ensemble average for the operand arg.

The corresponding single-run operator is

avg(arg) that compute the average from all elements of the (multi-dimensional) operand arg.

`avg(arg)` computes the average of arg from all of its elements for the default run number 0 of the experiment as it is applied outside a multi-run operator
Dimensionality = 0
Coordinates = (without)
Extents = (without)

`avg_e(arg)` computes for each element of arg its average over all single runs (excluding run no. 0) of the run ensemble
Dimensionality = dimensionality of arg
Coordinates = coordinates of arg
Extents = extents of arg

`avg_e(avg(arg))` at first computes for all single run individually from all values of arg their average (1 value per single run) and thereof the average over the whole run ensemble.
Dimensionality = 0
Coordinates = (without)
Extents = (without)

`avg(avg_e(arg))` at first computes the run ensemble average of arg that has the same dimensionality, coordinates and extents as the operand arg itself and afterwards computes the average from all values of the previous multi-run intermediate result
Dimensionality = 0
Coordinates = (without)
Extents = (without)

`avg(arg) + avg_e(arg) avg_e(avg(arg))+avg(avg_e(arg))` is allowed and is the sum of the four summands as described above
Dimensionality = dimensionality of arg
Coordinates = coordinates of arg
Extents = extents of arg

Example 8.9 *Multi and single run experiment post-processing operators*

8.4.2 Global Sensitivity Analysis – Elementary Effects Method GSA_EE

Tab. 8.9 *Experiment specific operators for GSA_EE*

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 118)	Argument value restriction
morris(arg1)	global sensitivity measures for argument arg	(6) $dim_{res} = dim_{arg1} + 2$ $ext_{res}(dim_{res}-1)$ = no._of_factors $coord_{res}(dim_{res}-1)$: name = factor_sequ values = equidist_end 1(1) no._of_factors $ext_{res}(dim_{res}) = 2$ $coord_{res}(dim_{res})$ = name = stat_measure values = equidist_end 4(1)5	
same as for UNC_MC	see Section 8.4.5	see Section 8.4.5	see Section 8.4.5

Having a model output variable definition as in Example 5.1 on page 44.
Assume the experiment description file GSA_EE from Example 6.3 on page 77
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_GSA_EE

Example 8.10 *Experiment post-processing operators for GSA_EE*

The following explanations hold for the operators in Tab. 8.10:

- The **operator morris** appends two additional dimensions to the dimensionality of its argument. The first corresponds to the 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. 11.11 the first index of this dimension holds the averages μ^{abs} and the second index the variances σ^2 to describe the importance of the corresponding factors. The sequence of the factors in the appended factor dimension corresponds to the sequence of the factors in <model>.edf.
- This experiment type allows to post-process the whole run ensemble as an UNC_MC experiment. Keep in mind that generally the sampled factor points do not follow a pre-defined distribution as they form special designed trajectories in the factor space.

8.4.3 Global Sensitivity Analysis – Variance-Based Method GSA_VB

Tab. 8.10 *Experiment specific operators for GSA_VB*

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 118)	Argument value restriction
effects_1st(arg1)	first order sensitivity indices of arg1	(6) $dim_{res} = dim_{arg1} + 1$ $ext_{res}(dim_{res})$ = no._of_factors $coord_{res}(dim_{res})$: name = factor_sequ values = equidist_end 1(1) no._of_factors	arg1 must not contain strings enclosed in ‘ ‘
effects_tot(arg1)	total effect sensitivity indices of arg1	(6) $dim_{res} = dim_{arg1} + 1$ $ext_{res}(dim_{res})$ = no._of_factors $coord_{res}(dim_{res})$: name = factor_sequ values = equidist_end 1(1) no._of_factors	arg1 must not contain strings enclosed in ‘ ‘
gsa_vb_run_mask(char_arg1) **this is NOT a multi-run operator!**	mask runs to use for subsequent UNC_MC experiment specific operators	(7)	sample+resample sample resample all resample_but_ <factor_name> resample_but_all
same as for UNC_MC	see Section 8.4.5	see Section 8.4.5	see Section 8.4.5

The following explanations hold for the operators in Tab. 8.10:

- For operators **effects_1st and effects_tot** first order and total effects (indices) are derived for reasons of numerical stability from the standardized variable (arg1 – avg_e(arg1)) / sqrt(var_e(arg1)) instead of arg1. This normalization does not change the indices. Both operators append an additional dimensions to the dimensionality of its argument. It corresponds to the factors. Their sequence corresponds to the sequence of the factors in <model>.edf.
- As the run ensemble is composed from the sample S, the re-sample R and the resulting samples RS_i (check Section 4.3) the operator **gsa_vb_run_mask** allows for selection of those sub-samples out of the run ensemble that are to be included into the performance of subsequent UNC_MC operators:

  If the character argument of the operator gsa_vb_run_mask is

  - sample+resample — all runs from the sample and the re-sample
  - sample — all runs from the sample
  - resample — all runs from the re-sample
  - all — all runs
  - resample_but_<factor_name> — all runs where all factors but <factor_name> are from the re-sample
  - resample_but_all — all runs but the sample and the re-sample

  are taken into account. The selected case is active until a new selection is made. The default value when starting the post-processor is set to sample+resample. Keep in mind that only the resulting runs of the first three cases follow the distributions as specified in <model>.edf by the sub-keyword *sample*. Generally, for the remaining cases the statistical measures of the factors (e.g., var_e) show a bias and this may be for measures of model output variables as well.
  The operator gsa_vb_run_mask is a stand-alone operator.

Having a model output variable definition as in Example 5.1 on page 44.
Assume the experiment description file GSA_VB from Example 6.4 on page 79
then in result-processing

```
gsa_vb_run('sample+resample')   set runs to use from the sample S and the re-sample R
                                for following post-processing with UNC_MC operators
effects_tot(atmo_g)             get total sensitivity indices for atmo_g
                                for the four defined factors
                                Dimensionality = 2
                                Coordinates = time , factor_sequ
                                Extents = 20 , 4
max_e(atmo_g)                   run ensemble maximum of atmo_g
                                from the sample S and the re-sample R
                                Dimensionality = 1
                                Coordinates = time
                                Extents = 20
sum(effects_1st(max(atmo)))     sum up linear sensitivity indices of max(atmo)
                                for a linearity test of this model output
                                Dimensionality = 0
                                Coordinates = (without)
                                Extents = (without)
```

Example file: world.post_GSA_VB

Example 8.11 *Experiment post-processing operators for GSA_VB*

8.4.4 Deterministic Factorial Design DFD

There is only one experiment specific operator for DFD experiment post-processing. With this operator dfd

- A single run can be selected from the run ensemble
- The complete run ensemble can be addressed
- Sub-spaces from the factor 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 dfd the simple experiment layouts as described in Fig. 4.6 on page 26 are used in the examples below.

- With the operator dfd it is possible to address for any operand a single run out of the run ensemble by fixing values of experiment factors. Dimensionality and extents of the operator result is the same as that of the operand.

  Example:
  for Fig. 4.6 (a) fix a value of factor p1 and of factor p2
  for Fig. 4.6 (b) fix a value of the parallel factors p1 or p2
  for Fig. 4.6 (c) fix values of factors p3 and p1 or p2

- Without any selection in the factor space the dimensionality of the operator result is formed from the dimensionality of the operand enlarged by the dimensionality of the factor space. The extents of the appended dimensions are determined by the number of sampled values.

  Example:
  for Fig. 4.6 (a) two additional dimensions are appended to the operand dimension
  for Fig. 4.6 (b) one additional dimension is appended to the operand dimension
  for Fig. 4.6 (c) two additional dimensions are appended to the operand dimension
  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.

- As a third option it is possible to select only a sub-space out of the factor space.

  Example:
  for Fig. 4.6 (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 experiment description file DFD_a in Example 6.5 in Section 6.4.2 on page 82.

- The operator dfd also enables to aggregate operands in the factor space. Sequence of performing aggregations is important.

  Example:
  for Fig. 4.6 (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 dfd.

Tab. 8.11 *Experiment specific operators for DFD*

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
dfd(char_arg1, arg2)	navigation and aggregation in the factor space for arg2 according to char_arg1	dim_{res} = dim_{arg2} + appended dimensions according to char_arg1 char_arg1= selection / aggregation filter according to Tab. 8.15	
same as for UNC_MC	see Section 8.4.5	see Section 8.4.5	see Section 8.4.5

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

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$_i$> = <val_int> for <factor_val_type> = i <val$_i$> = <val_float> else
<aggreg_type>	an aggregation / moment operator from Tab. 8.5 on page 123. 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 **dfd**:

- 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:

  Example: `dfd(' ',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>(*):

  Example:
  ```
  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 dfd 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 6.4.1) comb-line of the experiment description file is used in the dfd-result.
- Aggregation operators reduce dimensionality of the covered experiment factor space in the dfd-result. The sequence of aggregation operators in the first argument of the operator dfd influences the result: Computation starts with the first aggregation operator and ends with the last:

  Example: `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 dfd. 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 dfd 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 37) 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 dfd 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 dfd the factors that contribute to additional dimensions of the result of the operator dfd are appended to the dimensions of the operand arg2 of dfd according to the sequence they are used in the normalized (see Section 6.4.1) 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 6.4.1) comb-line is used in the result.

- For experiments that use a sample file (<model>.edf: specific comb file ...) instead of explicit sample definitions (<model.edf>: specific comb [default | <combination>]) all experiment factors are assumed to be combined in parallel.
- Additionally, this experiment type allows to post-process the whole run ensemble as an UNC_MC experiment. Keep in mind that generally the factors do not follow a pre-defined distribution.

Having a model output variable definition as in Example 5.1 on page 44 and
assuming address_default = coordinate in world_*.cfg.
Assume the experiment layout in Fig. 4.6 (c) on page 26 and
the corresponding experiment description file from Example 6.5 DFD_c on page 82
then in result-processing

```
dfd(' ',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

```
dfd('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

```
dfd('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

```
dfd('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

```
dfd('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

```
dfd('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)

```
dfd('max(p3),min(p2)',avg(atmo(*,*,1,19:20)))
```
Result differs normally from min(p2),max(p3)
(previous result expression)

```
dfd('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

```
dfd(' ',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_DFD_c

Example 8.12 *Experiment post-processing operator dfd for DFD*

8.4.5 Uncertainty Analysis – Monte Carlo Method UNC_MC

Tab. 8.13 shows experiment specific operators for an UNC_MC experiment that can be used in post-processing.

Tab. 8.13 *Experiment specific operators for UNC_MC (without standard aggregation / moment operators)*

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 118)	Argument value restriction
min_e(arg1)	run ensemble minimum	(1)	
max_e (arg1)	run ensemble maximum	(1)	
sum_e (arg1)	run ensemble sum	(1)	
avg_e (arg1)	run ensemble average	(1)	
var_e (arg1)	run ensemble variance	(1)	
avgg_e (arg1)	run ensemble geometric mean	(1)	
avgh_e (arg1)	run ensemble harmonic mean	(1)	
avgw_e (arg1, arg2)	run ensemble weighted mean of arg1	(2.1) arg2 = weight	

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 118)	Argument value restriction
hgr_e(char_arg1, int_arg2, real_arg3, real_arg4, arg5)	run ensemble heuristic pdf's separately calculated for each index of arg5	dim_{res} = dim_{arg5}+1 $ext_{res}(j)$ = $ext_{arg5}(j)$ for j = 1,…, dim_{arg5} $ext_{res}(dim_{res})$ = number of bins $coord_{res}(dim_{res})$ = char_arg1 for char_arg1 = 'bin_no' (bin number): $coord_{res}(dim_{res})$ values = equidist_end 1 (1) no._of_bins for char_arg1 = 'bin_mid' (bin mid): $coord_{res}(dim_{res})$ values = equidist_end 1st_bin_mid (bin_width) no._of_bins char_arg1 = ['bin_no' | 'bin_mid'] int_arg2 = number of bins 4 ≤ int_arg2 ≤ no._of_runs or = 0: automatic determination = max(4,no._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_e(arg5) and max_e(arg5) min_e(arg5) = max_e(arg5): all result values are undefined	
count_e(char_arg1, arg2)	run ensemble number of values	(1) char_arg1 = ['all' | 'def' | 'undef']	
maxprop_e(arg1)	return the run number where the maximum is reached the first time	(1) processing sequence starts with run number 1	
minprop_e(arg1)	return the run number where the minimum is reached the first time	(1) processing sequence starts with run number 1	
cnf_e (real_arg1, arg2)	run ensemble positive distance of confidence measure from avg_e(arg2)	(1) real_arg1 probability of error	real_arg1 = [0.001 | 0.01 | 0.05 | 0.1]
cor_e (arg1, arg2)	run ensemble correlation coefficient between arg1 and arg2	(2.1)	
cov_e (arg1, arg2)	run ensemble covariance between arg1 and arg2	(2.1)	
ens(arg1)	run ensemble values	(6) dim_{res} = dim_{arg1}+1 $ext_{res}(dim_{res})$ = no._of_runs $coord_{res}(dim_{res})$ = name = run values = equidist_end 1(1) no._of_runs	

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 118)	Argument value restriction
krt_e (arg1)	run ensemble kurtosis (4^{th} moment)	(1)	
med_e (arg1)	run ensemble median	(1)	
qnt_e (real_arg1, arg2)	run ensemble quantile of arg2	(1) real_arg1 quantile value	0. ≤ real_arg1 ≤ 100.
reg_e (arg1, arg2)	run ensemble linear regression coefficient to forecast arg2 from arg1	(2.1)	
rng_e (arg1)	run ensemble range = max_e(arg1) - min_e(arg1)	(1)	
skw_e (arg1)	run ensemble skewness (3^{rd} moment)	(1)	
stat_full(real_arg1, real_arg2, real_arg3, real_arg4, arg5)	run ensemble full basic statistical measures of arg5	(6) dim_{res} = dim_{arg5}+1 $ext_{res}(dim_{res})$ = 10 $coord_{res}(dim_{res})$ = name = stat_measure values = equidist_end 1(1)10	real_arg1, real_arg2 = [0.001 | 0.01 | 0.05 | 0.1] real_arg1 < real_arg2 probability of error for confidence distance measure 0. ≤ real_arg3 < real_arg4 ≤ 100. quantile values
stat_red(real_arg1, real_arg2, arg3)	run ensemble reduced basic statistical measures of arg3	(6) dim_{res} = dim_{arg5}+1 $ext_{res}(dim_{res})$ = 7 $coord_{res}(dim_{res})$ = name = stat_measure values = equidist_end 1(1)7	real_arg1, real_arg2 = [0.001 | 0.01 | 0.05 | 0.1] real_arg1 < real_arg2 probability of error for confidence distance measure

The following explanations hold for the operators in Tab. 8.13:

- All UNC_MC operators but ens, stat_full and stat_red have the postfix _e.
- All UNC_MC operators are also available for all other experiment types. Keep in mind that in general the samples from these experiment types do not follow any distribution.
- 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 but 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 and meaning is pre-defined by SimEnv (cf. Tab. 11.11).

  The values of the dimension generated additionally as the last argument by applying the operators stat_full and stat_red of the 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 a probability of error real_arg1
7. Run ensemble positive distance of confidence measure from run ensemble mean for a 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

Both operators were designed for application of an appropriate visualization technique in result evaluation in future. The operator stat_red was introduced as computation of the median and quantiles consumes a lot of auxiliary storage space.

Having a model output variable definition as in Example 5.1 on page 44 and
assuming address_default=coordinate in world_*.cfg.
Assume the UNC_MC experiment with the experiment description file UNC_MC from Example 6.6 on page 86
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_UNC_MC

Example 8.13 *Experiment post-processing operators for UNC_MC*

8.4.6 Local Sensitivity Analysis LSA

Tab. 8.14 shows the experiment specific operators for LSA that can be used in post-processing. For a definition of these operators check Tab. 4.5 on page 30.

Tab. 8.14 *Experiment specific operators for LSA*

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
sens_abs(char_arg1, arg2)	absolute sensitivity measure for arg2 according to char_arg1	dim_{res} = dim_{arg2} + appended dimensions according to char_arg1 char_arg1 = selection / aggregation filter	
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		
same as for UNC_MC	see Section 8.4.5	see Section 8.4.5	see Section 8.4.5

This experiment type allows to post-process the whole run ensemble as an UNC_MC experiment. Keep in mind that generally the factors do not follow a pre-defined distribution.

Tab. 8.15 *Syntax of the filter argument 1 for LSA*

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$_i$> = <val_int> for <val_type> = i <val$_i$> = <val_float> for <val_type> = s <val$_i$> = <factor_name> 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 the LSA 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 LSA 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 LSA 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. 11.11.
  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 44 and
assuming address_default=coordinate in <model>.cfg
Assume the experiment description file LSA from Example 6.7 on page 87
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_LSA

Example 8.14 *Experiment post-processing operators for LSA*

8.4.7 Bayesian Technique – Bayesian Calibration BAY_BC

Tab. 8.16 shows experiment specific operators for a BAY_BC experiment that can be used in post-processing.

The following explanations hold for the operators in Tab. 8.16:

- As discussed in Section 6.7.3, for points in the MCMC chain that are marked as outranged in the file <model>.blog a single simulation run is not performed. Consequently, they are not available from experiment output.
- For the multiple setting likelihood function case (check Section 6.7.2) only the run ensemble from an individual setting can be post-processed. Individual setting experiment outputs are stored in subdirectories "_s_<setting_number>" of the model output directory as specified in <model>.cfg. For post-processing adapt <model>.cfg accordingly.
- Operators **bay_bc_run_weight** and **bay_bc_run_weight**:
  Both operators are **not** multi-run post-processing operators. By definition, such operators can be applied outside (a multi-run operator is an argument to the operator) as well as inside (the operator is an argument of a multi-run operator) multi-run operators. In fact, both operators do not make sense when applying outside a multi-run operators. This is not checked when analysing a result expression.

- Operator **bay_bc_run_weight**:
  The weight that is assigned to a single run of type “accepted” or “reflected” is the number of all subsequent single runs of types “outrange” or “rejected” until the next single run of type “accepted” or “reflected”. The single run under consideration also contributes to the weight. The resulting weight is multiplied with arg1. Single runs of type “rejected” get an undefined weight.
- This experiment type allows to post-process the whole run ensemble as an UNC_MC experiment.

Tab. 8.16 *Experiment specific operators for BAY_BC*

Name	Meaning	Argument restriction(s) / result description (Tab. 8.1, page 118)	Argument value restriction
bay_bc_run_mask(arg1) **this is NOT a multi-run operator!**	mask all values of arg1 (set them undefined) that belong to single runs of type “rejected”	(1)	arg1 must not be a multi-run operator
bay_bc_run_weight(arg1) **this is NOT a multi-run operator!**	mask all values of arg1 (set them undefined) that belong to single runs of type “rejected” and weight arg1 of unmasked values (values that belong to single runs of type “accepted” or “reflected”)	(1)	arg1 must not be a multi-run operator
same as for UNC_MC	see Section 8.4.5	see Section 8.4.5	see Section 8.4.5

Mask rejected runs of the whole run ensemble:

```
ens(bay_bc_run_mask(arg))          returns for rejected runs the undefined value,
                                   otherwise arg
run('1',bay_bc_run_mask(arg))      returns for run no. 1 the undefined value if it was
                                   rejected, otherwise arg
bay_bc_run_mask(ens(arg))          returns a meaningless result
```

Mask rejected runs and weight runs of type accepted or reflected of the whole run ensemble:

```
ens(bay_bc_run_weight(arg))        rejected runs get undefined weights, accepted and
                                   reflected runs get a weight which is the number of
                                   assigned subsequent runs of type outrange or
                                   rejected. Sum is icreased by 1.
                                   Weight is multiplied with arg.
sum e(bay bc run weight(1)) + run('0',bay bc run weight(1))-1
                                   the result returns the chain length
```

Example file: world.post_BAY_BC

Example 8.15 *Experiment post-processing operators for BAY_BC*

8.4.8 Optimization – Simulated Annealing OPT_SA

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. 8.17 *Experiment specific operators for OPT_SA*

Name	Meaning	Argument restriction(s) / result description	Argument value restriction
same as for UNC_MC	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 an UNC_MC experiment. Keep in mind that the factors generally do not follow a pre-defined distribution.

Having a model output variable definition as in Example 5.1 on page 44 and
assuming address_default=coordinate in <model>.cfg
Assume the experiment description file LSA from Example 6.12 on page 97
then in result-processing

`minprop_e(-sum(bios))`
run number with the minimal cost function
Dimensionality = 0
Coordinates = (without)
Extents = (without)

`ens(-sum(bios)-run('xxx',-sum(bios)))`
run ensemble of the bias of cost function from its optimal value. Replace **xxx** by the result of the previous operator.
Dimensionality = 1
Coordinates = run
Extents = 500

`rank('tie_min',ens(-sum(bios)))`
run ensemble of the ranks of the cost function
Dimensionality = 1
Coordinates = run
Extents = 500

Example file: world.post_OPT_SA

Example 8.16 *Experiment post-processing operators for OPT_SA*

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 simulation 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 with pre-defined names in C/C++ or Fortran that represent the check and the computational part of the user-defined operator. 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.

User-defined operators must not be multi-run operators. Consequently, such operators can not process the whole run ensemble according to any processing strategy for the individual experiment type. For example, for experiment type UNC_MC, it is impossible to declare a user-defined operator that derives any measure from the whole run ensemble. However, the concept of composed operators allows for such an approach by applying built-in multi-run operators.

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. 8.18):

- 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.

Tab. 8.18 *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, len_res)	compute result of the operator in dependency on operands	real*4 res(len_res)	(output) result vector of the operator
		integer*4 len_res	(input) length of the 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 array res using the Fortran storage model (cf. Section 15.7 – Glossary).**

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. 8.19 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. 8.19 *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 sufficient 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
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

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
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*64 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 **Application of this function in simenv_check_ user_def_operator for coordinate bin_mid results in an error**	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]
		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

Function name	Function description	Inputs / outputs / function value	Inputs / outputs / function value description
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 in-place 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. 8.20 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 advisable 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. 8.20 *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 column-major order 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 files simenv_opr_f.inc and simenv_opr_c.inc from the inc subdirectory 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. 11.6).
- Apply the shell script

  link_simenv_opr_[f | c | cpp].sh <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. 15.14 lists the additionally used symbols when linking a user-defined operator.
- In Section 15.3 on page 241 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. 8.20) 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_[f | c]( ) (cf. Tab. 8.20)
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        character       "normal"        composed operator
operator name   argument        argument        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 44
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 8.17 *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: (related to Example 8.17)
  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: (related to Example 8.17)
  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 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 12.1 on page 203 with the keywords, names, sub-keywords, and values as in Tab. 8.21. <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 evaluated to check a user-defined and/or composed operator syntactically when performing it during experiment post-processing.

Tab. 8.21 *Elements of an operator description file <model>.odf (line type: m = mandatory, o = optional)*

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. 8.21 the following additional rules and explanations apply:

- 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 8.18 *Operator description file <model>.odf*

8.6 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 8.19 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 12.1 on page 203 with the keywords, names, sub-keywords, and values as in Tab. 8.22. <model>.mac describes the user-defined macros.

Tab. 8.22 *Elements of a macro description file <model>.mac (line type: m = mandatory, o = optional)*

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).

To Tab. 8.22 the following additional rules and explanations apply:

- Values for sub-keywords *descr* and *unit* are not evaluated during parsing a result expression.

```
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 8.19 *User-defined macro definition file <model>.mac*

8.7 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 and the corresponding optional result descriptor (for both check Section 8.1.1) 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. The same applies to the optional result descriptor.

Wildcard operands must not be used in macro definitions (cf. Section 8.6). 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 44 and assuming the experiment description file DFD_b from Example 6.5 on page 82 then in result-processing

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

Example 8.20 *Experiment post-processing with wildcard operands*

8.8 Undefined Results

By performing operator chains and due to possibly undefined / 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 Section 11.8.

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 SimEnvVis developers on request.

Analysis and evaluation of post-processed data selected and derived from large amount of relevant experiment 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 preformed visualization modules are evaluated by a suitability coefficient how they can map the data in an appropriate manner.
The SimEnvVis framework offers visualization modules with 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 coordinates, 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 adapted, 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 11.2) and directly during experiment post-processing by the service simenv.res if in the file <model>.cfg (check Section 11.1) 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 11.2 for more information.

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

10 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. Other 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 files

<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 column-major order model (cf. Section 15.7 – Glossary).
Use the SimEnv service command simenv.dmp for browsing model and result output files. See Tab. 11.4 for more information.

10.1 NetCDF Model and Experiment Post-Processor Output

The intention for applying NetCDF format for model and experiment post-processor output is to provide a self-describing, platform-independent data file format with metadata that can be interpreted by subsequent analyses techniques such as visualization. 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 1.6 (LLNL, 2012). Additionally, global and local attributes – the latter in NetCDF terminology for independent and dependent NetCDF variables – that are to be deleted, replaced or defined are described by the file <model>.ndf (see Section 10.1.3).

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 (cf. Tab. 5.4) are automatically transferred into NetCDF data types (cf. the Table below). By default, all post-processor output data is of type float / real*4.

Tab. 10.1 *NetCDF data types*

SimEnv data type (cf. Tab. 5.4)	NetCDF data type
byte / int*1	NF_BYTE
short / int*2	NF_SHORT
int / int*4	NF_INT
float / real*4	NF_FLOAT
double / real*8	NF_DOUBLE

10.1.1 Global Attributes

The global attributes used in SimEnv from the NetCDF-CF standard for model and post-processor output are :Conventions, :institution, and :title. They are complemented by CF non-compliant attributes that are all prefixed by “SimEnv_”

Tab. 10.2 *Global NetCDF attributes*

Name	Value	Data type
:Conventions	CF-<CF_version>	char
:institution	as specified in $SE_HOME/bin/simenv_settings.txt	char
:title	SimEnv Vers. <SimEnv_version> [model | post-processor] output # Workspace: <user_name>@<hostname>:<SimEnv_workspace> # Timestamp: <YYYY-MM-DD>T<hh:mm:ss>Z # SimEnv: http://www.pik-potsdam.de/software/simenv/	char
:SimEnv_Conventions (NetCDF-CF non-compliant)	information on CF convention extensions	char
:SimEnv_model (NetCDF-CF non-compliant)	<model>: <model description from “ general descr “ of <model>.mdf>	char
:SimEnv_experiment (NetCDF-CF non-compliant)	<xxx> single runs with experiment type <acronym and long name – see Tab. 4.1>: <experiment description from “ general descr “ of <model>.edf>	char

10.1.2 Variable Labeling and Variable Attributes

For NetCDF variables, two cases of labeling 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 is used.
- Otherwise, an additional coordinate

  <var_name>-<co_name>

  is defined, where the NetCDF variable depends on. The additional coordinate is described in the dimension and data part of the NetCDF file. Additionally, the SimEnv specific attribute

  index_range_<original_coordinate_name> (see Tab. 10.3)

  is assigned to such a NetCDF variable. Check also Example 10.1 and Example 10.2.

Having a model output variable definition as in Example 5.1 on page 44 and assuming in <model>.cfg

```
model     out_format        netcdf
model     out_separation    yes
```

During experiment performance model output variables atmo and bios are stored as follows: (excerpt from ncdump)

```
netcdf world_f.out000000 {
dimensions:
        lat = 45 ;
        lon = 90 ;
        level = 4 ;
        time = 20 ;
        bios-lat = 36 ;
variables:
    float lat(lat) ;
        lat:standard_name = "SimEnv model coordinate" ;
        lat:long_name = "geographic latitude" ;
        lat:units = "deg" ;
        lat:axis = "Y" ;
    …
    float atmo(time, level, lon, lat) ;
        atmo:standard_name = "SimEnv model output variable" ;
        atmo:long_name = "aggregated atmospheric state" ;
        atmo:units = "atmo_unit" ;
        atmo:_FillValue = 3.4e+38f ;
        atmo:data_range = -1999.95f, 2002.39f ;

    float bios-lat(bios-lat) ;
        bios-lat:standard_name = "SimEnv model coordinate" ;
        bios-lat:axis = "Y" ;

    float bios(time, lon, bios-lat) ;
        bios:standard_name = "SimEnv model output variable" ;
        bios:long_name = "aggregated biospheric state" ;
        bios:units = "bios_unit" ;
        bios:index_range_lat = 2, 37 ;
        bios:_FillValue = 3.4e+38f ;
        bios:data_range = -1982.372f, 1999.391f ;
    …
lat =    88, 84, …, -84, -88 ;
    …
bios-lat = 84, 80, …, -52, -56 ;
    …
```

Example 10.1 *Additional coordinates in NetCDF model output*

Having a model output variable definition as in Example 5.1 on page 44 and assuming in <model>.cfg

```
postproc     out_format      netcdf
```

During experiment post-processing results atmo and run('0',sin(atmo(i=2:37,*,*,*))) are stored as follows: (excerpt from ncdump)

```
netcdf world_f.res01 {
dimensions:
        lat = 45 ;
        lon = 90 ;
        level = 4 ;
        time = 20 ;
        res_02-lat = 36 ;
variables:
        float lat(lat) ;
            lat:standard_name = "SimEnv model coordinate" ;
            lat:long_name = "geographic latitude" ;
            lat:units = "deg" ;
            lat:axis = "Y" ;
    …
        float res_01(time, level, lon, lat) ;
            res_01:standard_name = "aggregated atmospheric state" ;
            res_01:long_name = "atmo" ;
            res_01:units = "atmo_unit" ;
            res_01:data_range = -1999.95f, 2002.39f ;
            res_01:_FillValue = 3.4e+38f ;

        float res_02-lat(res_02-lat) ;
            res_02-lat:standard_name = "SimEnv model coordinate" ;
            res_02-lat:long_name = "geographic latitude" ;
            res_02-lat:axis = "Y" ;
            res_02-lat:units = "deg" ;

        float res_02(time, level, lon, res_02-lat) ;
            res_02:standard_name = "aggregated atmospheric state" ;
            res_02:long_name = "sin(atmo(i=2:37,*,*,*))" ;
            res_02:units = "atmo_unit" ;
            res_02:data_range = -1.f, 1.f ;
            res_02:index_range_lat = 2, 37 ;
            res_02:_FillValue = 3.4e+38f ;
    …
lat =          88, 84, …, -84, -88 ;
    …
res_02-lat =   84, 80, …, -52, -56 ;
    …
```

Example 10.2 *Additional coordinates in NetCDF post-processor output*

The following NetCDF variable attributes are used according to the NetCDF-CF standard.

Tab. 10.3 *Variable NetCDF attributes*

Name	Value	Data type
NetCDF-CF compliant		
<NetCDF_variable_name> :standard_name	see below	char
<NetCDF_variable_name> :long_name	see below	char
<NetCDF_variable_name> :units	see below	char
<NetCDF_variable_name> :axis	[X | Y | Z | T] (only for <NetCDF_variable_name> = <co_name>)	char
<NetCDF_variable_name> :_FillValue	variable type-depending missing value – see Tab. 11.13	type-dep.
NetCDF-CF non-compliant		
<NetCDF_variable_name> :data_range	<value_min> <value_max>	type-dep.
<NetCDF_variable_name> :index_range_<coordinate>	<index_min> <index_max>	int

To Tab. 10.3 the following rules and explanations apply:

- For the **:standard_name**, **:long_name** and **:units** attributes the following information from the optional result descriptor of an result (see Section 8.1.1) and from <model>.mdf, <model>.edf are used. Keep in mind that the :standard_name attribute value normally does not follow the NetCDF-CF standard as it is defined in the optional result descriptor, <model>.edf, or <model>.mdf and is not checked for CF compliance.
  - For model output:

    If <NetCDF_variable_name> = <var_name> or built-in variable name (see Tab. 11.9)
    then :long_name = 'SimEnv {built-in} model output variable'

    If <NetCDF_variable_name> = <factor_name>
    then :long_name = 'SimEnv model factor'

    For both types is

    :standard_name = <descr_string> (if specified)
    (Attribute is omitted) (otherwise)
    :units = <unit_string> (if specified)
    (otherwise)

    where <descr_string> und <unit_string> are the related strings for <var_name> from <model>.mdf and/or for <factor_name> from model.edf
  - Post-processor output:

    <NetCDF_variable_name> = <result_name> (if defined in result descriptor)
    res<digit><digit> (otherwise)
    :standard_name = <result_description> (if defined in result descriptor and/or <model>.mdf; see below)
    (Attribute is omitted) (otherwise)

:long_name =	<result_expression> 'SimEnv post-processor output chain'	(see Section 8.1.1, macros are resolved)
:units =	<result_unit>	(if defined in result descriptor and/or <model>.mdf; see below)

Check Section 8.1.1 for more information on the approach how missing information on <result_description> and <result_unit> from the optional result descriptor can be emulated from the corresponding information from <model>.mdf for single operand – single operator result expressions.

- For coordinates in model output or post-processor output:

  If <NetCDF_variable_name> = <co_name> or [var_name> | res_<digit><digit>]-<co_name> (see above)

then	:standard_name =	<descr_string>	(if specified in <model>.mdf)
		(Attribute is omitted)	(otherwise)
	:long_name =	'SimEnv model coordinate'	
	:units =	<unit_string>	(if specified in <model>.mdf)
		' '	(otherwise)

  where <descr_string> und <unit_string> are the related strings for <co_name> / [<var_name> | res_<digit><digit>]-<co_name> from <model>.mdf or a built-in coordinate name (see Tab. 11.11)

- The **:axis** attribute is assigned to a coordinate variable and used according to the NetCDF-CF standard to identify coordinates that correspond to latitude, longitude, vertical, or time axes. If the lowercase representation of a coordinate name <co_name> or [<var_name> | res_<digit><digit>]-<co_name> (for the latter see above) is
  - 'longitude' or 'lon' then :axis = 'X'
  - 'latitude' or 'lat' then :axis = 'Y'
  - 'level' or 'lev' or 'height' then :axis = 'Z'
  - 'time' or 'date' then :axis = 'T'

  For all other cases the :axis attribute is omitted.
- The **:_FillValue** attribute holds SimEnv type dependent undefined values according to Tab. 11.13. As result variables res_<digit><digit> are always of type float / real*4, the corresponding SimEnv undefined value is used. The _FillValue is not applied to coordinate variables but coordinate 'run'.
- The **:data_range** attribute provides the value range that is covered by the variable in the NetCDF file. The value range is bounded by the minimum and the maximum value of the variable.
  The visualization system SimEnvVis (see Chapter 9) that is coupled to SimEnv expects that the nodata value (see Section 11.8) assigned to the variable data type is outside the value range of this variable. Normally, type dependent default nodata values are outside the value range. This may be violated for user-defined nodata values. In particular, pay attention to a user-defined float / real*4 nodata type when post-processor output is stored in NetCDF format as post-processor NetCDF output files are visualized by SimEnvVis and all post-processor output is of type float / real*4.
- 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 as [<var_name> | res_<digit><digit>]-<co_name> (for the latter see above) to this NetCDF variable. The index_range attribute describes the sub-range of the coordinate <co_name> for which the NetCDF variable is defined. Range indices count from 1.

10.1.3 NetCDF Attribute Description File <model>.ndf

The optional SimEnv NetCDF attribute description ASCII file <model>.ndf allows to delete / insert and/or replace global and local (= NetCDF variable) attributes during experiment performance and result post-processing from / in the respective NetCDF output files. SimEnv NetCDF file based experiment performance, post-processing analysis, and the coupled visualization system SimEnvVis work with the global and local attributes as defined in Sections 10.1.1 and 10.1.2, respectively. Other analysis software with input from SimEnv NetCDF output may require modified or additional information / attributes. The file <model>.ndf easily and flexibly facilitates basic management of NetCDF attributes. Advanced functionality for NetCDF management is provided by open source tools such as the CDO (Schulzweida *et al.*, 2009) or NCO (NCO, 2012) operators.

<model>.ndf is an ASCII file that follows the coding rules in Section 12.1 on page 203 with the keywords, names, sub-keywords, and value as in Tab. 10.4. Use the SimEnv service simenv.chk to check a NetCDF attribute description file.

Tab. 10.4 *Elements of a NetCDF attribute description file <model>.ndf (line type: o = optional)*

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
general	<nil>	descr	o	any	<string>	general attribute description
glob_attr	<nil>	delete	o	1	<attrib_name>	delete global attribute <attrib_name>
		replace	o	any	<attrib_name> <string>	replace / insert global attribute <attrib_name> by <string>
loc_attr	[<var_ name> | <co_name> | &v& | &c& | &r& | &*&]	delete	o	1	<attrib_name>	delete local (NetCDF variable) attribute <attrib_name> for <var_name>, <co_name> or wildcard &...&
		replace	o	1	<attrib_name> [text <string> | [byte | short | int | float | double] <val_list>]] <val_list> = list <val_1>,…,<val_n>	replace / insert local (NetCDF variable) attribute <attrib_name> by <string> and/or <val_list> for <var_name>, <co_name> or wildcard &...&

To Tab. 10.4 the following additional rules and explanations apply:

- **Attribute names** <attr_name> and character strings <string> are case sensitive. Attribute names are not checked for NetCDF-CF compliance.
- The meaning of the **wildcard &...&** are as follows:
  - &v&: all variables (only active for model output)
  - &c&: all coordinates
  - &r&: all results (only active for post-processor output)
  - &*&: &c& and &v& (model output) and/or
    &c& and &r& (post-processor output)
- By the sub-keyword ***replace*** the specified attribute is replaced (and not appended!) if the attribute is already defined, otherwise it is inserted.
- If the sub-keywords ***delete*** and ***replace*** are declared for the same attribute name <attrib_name> and for a local attribute for the same <var_name> or <co_name> or wildcard &…& then the attribute is deleted first by 'delete' and afterwards inserted by 'replace'. Consequently, for such constellations delete is a redundant entry.
- If for a local attribute a **coordinate** <co_name> is addressed then also all derived coordinates <var_name>-<co_name> (see Section 10.1.2) are handled by this entry.
- For a local attribute <attrib_name> and multiple declarations for <var_name> / <co_name> and wild-card &…& the **processing sequence** is as follows independently on the declaration sequence in <model>.ndf:
  - &*&
  - &c& and &v& or &r&
  - <var_name> or <co_name>

  and within each processing step if specified, first for 'delete' and afterwards for 'replace'.

```
loc attr  &*&      replace  <attr name>       text <string *>
to define for each variable <var_name> and each coordinate <co_name>
an attribute <attr_name> with the value <string_*>

loc_attr   &v&           replace     <attr_name>    text <string_v>
afterwards to overwrite for each variable <var_name> the attribute <attr_name>
by the value <string_v>

loc_attr   <var_name1>   replace     <attr_name>    text
                                                    <string_var_name1>
afterwards to overwrite for variable <var_name1> the attribute <attr_name>
by the value <string_var_name>

loc_attr   <var_name2>   delete      <attr_name>
and to delete for variable <var_name2> the attribute <attr_name>
```

Example 10.3 *Processing sequence of the NetCDF attribute definition file*

- All global attribute values are of **data type** character, which corresponds to the identifier 'text' in <model>.ndf. Local attributes allow besides character attributes for numerical data type attributes, identified by 'byte', 'short', 'int', 'float' and/or 'double'. Numerical values of the corresponding type are declared after the type identifier in an explicit value list (see Tab. 12.5) as

  <val_list> = list <val_1>,…,<val_n>
- For additional examples check Example 10.4.

Having a model output description file as in Example 5.1 and an experiment description file DFD_a as in Example 6.5: then a NetCDF attribute definition file world.ndf could look like:

```
        # replace global attribute SimEnv_model:
glob_attr         replace   SimEnv_model   text  my own model description
        # replace in model output for factor p1 text attribute units:
loc_attr   p1     replace   units          text  new_unit
        # define in model output for all variables text attribute new_attr_1:
loc_attr   &v&    replace   new_attr_1     text  new attribute 1 text
        # delete in model output for variable atmo text attribute new_attr_1:
loc_attr   atmo   delete    new_attr_1
        # define in model output for variable bios float attribute new_attr_2 with 3 values:
loc_attr   bios   replace   new_attr_2     float list 3.14,1.e-12,4711
        # delete in post-processor output for all results text attribute standard_name:
loc_attr   &r&    delete    standard_name
```

Example 10.4 *NetCDF attribute definition file world.ndf*

10.2 IEEE Compliant Binary Model Output

IEEE compliant binary model output is written in records of fixed length to files

- <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. 11.9
- 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 according to the column-major order 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 (nominal) 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 DFD_a as in Example 6.5 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 int(1.555.312 : 512.000) +1 = 4 records with a fixed length of 512.000 Bytes. Remaining bytes in the last record are undefined.

Example 10.5 *IEEE compliant model output data structure*

10.3 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:

Tab. 10.5 *Record structure of <model>.inf<simenv_res_char>.[ieee | ascii] for each result*

Record #	Record length	Record contents
File header		
1	33+max. 10 characters	used SimEnv Version
2	33+max. 465 characters	user workspace, prefixed by user name and login node
3	33+20 characters	timestamp
Result number 1		
4	33+max. 1024 characters	result name string: same as the NetCDF variable name for post-processor output (Section 10.1.2)
5	33+max. 128 characters	result description string: same as NetCDF attribute :standard_name for post-processor output (Section 10.1.2)
6	33+max. 1024 characters	result operator string: same as NetCDF attribute :long_name for post-processor output (Section 10.1.2)
7	33+max. 32 characters	result unit string: same as NetCDF attribute :units for post-processor output (Section 10.1.2)
8	33 characters+9 int values	extents ext(1) ... ext(dim) 0 ... 0 of the result
9	33 +max. 64 characters	coordinate name of dimension 1
10, …	10 float values	ext(1) coordinate values of dimension 1 in records of 10 values (last record may have less values)
...		
xxx	33+max. 64 characters	coordinate name of dimension dim
xxx+1, …	10 float values	ext(dim) coordinate values of dimension dim in records of 10 values (last record may have less values)
Result number 2, …		
…		

The 33 characters prefixed in most of the records hold information on what the contents of the record is.

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 column-major order model (cf. Section 15.7 – Glossary).

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.

11 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.

11.1 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. 11.1 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. While all the entries in the table above and including “SimEnvVis_availability” are mandatory for any SimEnv installation, all entries below “SimEnvVis_availability” are only requested if SimEnvVis is available for / reachable from the current SimEnv installation.

Tab. 11.1 *Elements of the file simenv_settings.txt*

Keyword	Value	Explanation
institution	character string	institution SimEnv is used by (used in NetCDF output in global attribute "institution")
SimEnv_admin (*)	email address	email address of the SimEnv administrator
logfile_directory	<directory>	directory to store SimEnv log files (which services were used, …)
SimEnv_home_directory2log (*)	<directory>	SE_HOME directory to store log files from
drm_system	[LoadL | PBS/Torque]	installed distributed resource manager
drm_login_node (*)	hostname	login node for a compute cluster to access the
SimEnvVis_availability	[yes | no]	availability of visualization tool SimEnvVis
SimEnvVis_server_hostname	hostname	name of the visualization server that hosts the visualization component SimEnvVis
SimEnvVis_home_directory	<directory>	SimEnvVis home directory
SimEnvVis_working_directory	<directory>	SimEnvVis working directory on the visualization server
server_SimEnv_home_directory	<directory>	SE_HOME directory on the visualization server
ssh_local	<directory>/binary	ssh implementation of the client computer

scp_local	<directory>/binary	scp implementation of the client computer
ssh-keygen_local	<directory>/binary	ssh-keygen implementation of the client computer
nc_libpath	<directory>	directory the NetCDF shared object libraries are located for Linux

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 12.1 on page 203 with the keywords, names, sub-keywords, and info as in Tab. 11.2.

Tab. 11.2 *Elements of a general model-related configuration file <model>.cfg (line type: o = optional)*

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
		nodata_value_byte	o	1	<byte_val>	nodata value for integer*1 / byte data
		nodata_value_short	o	1	<short_val>	nodata value for integer*2 / short data
		nodata_value_int	o	1	<int_val>	nodata value for integer*4 / int data
		nodata_value_float	o	1	<float_val>	nodata value for real*4 / float data
		nodata_value_double	o	1	<double_val>	nodata value for real*8 / double data
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
		out_builtin_vars	o	1	[yes | no]	indicates whether built-in model variables (check Tab. 11.9) are stored to SimEnv model output files
		slices	o	1	[no | f_c | py_ja_m]	indicates whether simenv_slice_* is not used, used for Fortran / C or Python / Java / Matlab models
		structure	o	1	[standard | distributed | parallel]	indicates model structure with respect to experiment performance

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
experiment	<nil>	restart_ini	o	1	[no | yes]	perform <model>.ini for experiment restart
		begin_run	o	1	<val_int>	begin single run number
		end_run	o	1	[last | <val_int>]	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 values are stored in SimEnv model output
		display_values	o	1	<val_int>	specify number of values per result that are displayed additionally
		visualization	o		[yes | no]	determine whether to directly visualize an entered result during experiment post-processing

To Tab. 11.2 the following additional rules and explanations apply:

- 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. During running an experiment the values info and warning imply that the model output is checked on undefined values. This may result in a noticeable increase of CPU time consumption (see Section 11.3).
- For keyword *general*, sub-keywords ***nodata_value_byte, nodata_value_short nodata_value_int, nodata_value_float, nodata_value_double***:

  Check Section 11.8

- 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 set the value to no. If it is applied in Fortran or C/C++ models set the values to f_c and for Python, Java or Matlab model 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 = 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 experiment types UNC_MC with a stopping function, BAY_BC and OPT_SA 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 7.3.3 on page 104). 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 values 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 non-negative 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 non-negative 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*.
  The runs the experiment will be performed for are defined as follows:
  (i) get the intersection of the runs as specified by the experiment definition in <model>.edf and the include set
  (ii) form the relative complement of this set with respect to 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 ***display_values***:
  Enables display of a number of result values per result during experiment post-processing. For multidimensional result output the first values according to the column-major order model (cf. Section 15.7 – Glossary) are shown. Set display_values to 0 to suppress this output.
- For keyword *postproc*, sub-keyword ***visualization***:
  Specifies whether to directly visualize an entered result during experiment post-processing. Works only if the visualization system SimEnvVis is available according to the value of the entry SimEnvVis_availability in Tab. 11.1.

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. 11.3 lists the default values in the general configuration file in the case of absence of the appropriate sub-keyword.

Tab. 11.3 *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
	nodata_value_byte	127	Section 11.8
	nodata_value_short	32767	Section 11.8
	nodata_value_int	2147483647	Section 11.8
	nodata_value_float	3.40E+38	Section 11.8
	nodata_value_double	1.79D+308	Section 11.8
model	out_directory	./	above
	out_format	NetCDF	Chapter 10
	out_separation	yes	above
	out_builtin_vars	yes	Tab. 11.9
	slices	no	above
	structure	standard	Section 5.11 and above
experiment	restart_ini	no	Section 7.4
	begin_run	0	Chapter 7 and above
	end_run	last	Chapter 7 and above
	include_runs	<nil>	above
	exclude_runs	<nil>	above
	email	<nil>	Section 7.1
postproc	out_directory	./	above
	out_format	NetCDF	Chapter 10
	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
	display_values	0	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 11.1 *User-defined general configuration file <model>.cfg*

11.2 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 general SimEnv log files and to wrap up a SimEnv workspace.

Tab. 11.4 *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 SimEnvVis runs on the server viss.

SimEnv service	Use to
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, experiment 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 column-major order model for multi-dimensional fields (cf. Section 15.7 – Glossary).
simenv.hlp <topics>	acquire basic SimEnv **help** information for the specified topics
simenv.sta <user_name> {<begin_date>} {<end_date>} {<sort>}	generate general 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.key <user_name>	get 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.

To Tab. 11.4 the following remarks and explanations apply:

- All but services simenv.cpy, simenv.hlp, simenv.key, and simenv.sta:
  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.

11.3 Experiment Performance Tuning

SimEnv allows for a bunch of settings that influence in terms of CPU time the performance of the single simulation runs and their communication with the simulation environment. In particular, these are

- **The model output format**
  as specified in the general configuration file <model>.cfg by
  model out_format [netcdf | ieee]
  Model output to compliant IEEE format is faster than to NetCDF format.
  Nevertheless, NetCDF comes with all the advantages of a self-describing data format.

- **A lumped experiment output into a common file or into an individual model output file for each single run**
  as specified in the general configuration file <model>.cfg by
  model out_separation [yes | no]
  Experiment output to individual model output files is faster than to a lumped file.
  Nevertheless, for a moderate model output size per single run a compact output into a lumped file is more convenient.

- **Using slices**
  as specified in the general configuration file <model>.cfg by
  model slices [no | f_c | py_ja_m]
  and in the interfaced model source code by
  the SimEnv interface function simenv_slice_*
  Not applying slices is faster than applying them for Fortran or C/C++ models. Compared to Fortran or C/C++ models applying slices for Python, Java or Matlab models increases model run time significantly.
  Nevertheless, for models that store a multi-dimensional model output field in a lower-dimensional array this is the only approach to allow for addressing the field by one output variable in SimEnv post-processing.

- **Checking model output on undefined values**
  As specified in the general configuration file <model>.cfg by
  general message_level [info | warning | error]
  Not checking model output on undefined values by setting the message level to error is faster than checking it. In particular, checking NetCDF model output results in a noticeable CPU time consumption.
  Nevertheless, checking on undefined values can support model debugging.

11.4 Model Interface Scripts, Include Files, Link Scripts

Tab. 11.5 lists all these dot scripts and shell scripts that can / must be used in <model>.[ini | run | end].

Tab. 11.5 *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. 11.10
***(*)**: 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

Dot script	Use status	Used for	See Section
<model>.run			
simenv_ini_sh	mandatory	init for any model	5.8
simenv_end_sh	mandatory	end 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_bay_bc_sh **(*)**	optional	control multiple setting case for a BAY_BC experim.	7.2
simenv_kill_process **(*)**	optional	kill a program / model after reaching a CPU time limit	6.7.2
<model>_[sh | as].inc	optional	semi-automated model interface at shell script / ASCII level (cf. also Tab. 11.6)	5.10
<model>.end			
simenv_end_gams	mandatory	experiment end for GAMS models	5.7

In Tab. 11.6 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. 11.6 *SimEnv include files and link scripts*

File / location	Used in / generated during	Explanation
link_simenv_mod_ [f | c | cpp].sh $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
link_simenv_opr_ [f | c | cpp].sh $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 restart, 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)

11.5 User-Defined Files and Shell Scripts, Temporary Files

Tab. 11.7 lists the mandatory or optional shell scripts and files the user has to provide for running SimEnv services.

Tab. 11.7 *User files and shell scripts to perform any SimEnv service*
(*)*: make sure by the Unix / Linux command chmod u+x <file>*
that the 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	11.1
<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	8.6
<model>.odf	ASCII user-defined operator description file	optional	8.5.4
<model>.bdf	ASCII user-defined Bayesian calibration description file	BAY_BC: mandatory	6.7.1
<model>.gdf	ASCII user-defined GAMS model output description file	GAMS models: mandatory	5.7.2
<model>.ndf	ASCII user-defined NetCDF attribute description file	optional	10.1.3
<model>.run **(*)**	model shell script to wrap the model executable	mandatory	7.5
<model>.rst **(*)**	model shell script to prepare single model run restart	optional	7.5
<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	7.5
<model>.end **(*)**	model shell script to wrap up simulation experiment	optional, for GAMS models mandatory and standardized	7.5
l<model>.lnk **(*)**	model shell script to link an interfaced C/C++/Fortran model. Used in the course of experiment preparation for experiment run (not restart) 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/link_simenv_mod_[f | c | cpp].sh	optional	5.10
<model>_ [dis | par | seq]_ [aix | linux]. [jcf | pbs]	user-specific job control file to perform a model under Distributed Resource Manager control in distributed / parallel / sequential mode (jfc for LoadL, pbs for PBS/Torque)	optional	7.5
<model>_opt_sa_ options.txt	user-specific control and option file for experiment type OPT_SA	optional	6.8.1

Shell script / file (in the current workspace $SE_WS)	Explanation	Exist status	For more information see Section
<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	7.5
usr_opr_<opr> **(*)** (in the opr_directory according to <model>.cfg)	executable for user-defined operator <opr>	optional	8.5

Tab. 11.8 lists the temporary or permanent files that are created during a SimEnv service.

Tab. 11.8 *Files generated during performance of SimEnv services*
For the current workspace $SE_WS see Tab. 11.14.

File / location	Generated in	Explanation
Permanent files		
<model>.smp $SE_WS	experiment preparation (all but OPT_SA, BAY_BC) experiment performance (OPT_SA, BAY_BC)	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

File / location	Generated in	Explanation
<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>.blog $SE_WS	experiment performance (only for experiment type BAY_BC)	ASCII minutes file of Bayesian calibration
<model>.bmlog $SE_WS	experiment performance (only for experiment type BAY_BC)	ASCII minutes file of Bayesian calibration for the multiple setting case
<model>.olog $SE_WS	experiment performance (only for experiment type OPT_SA)	ASCII minutes file of **o**ptimization experiment performance
<model>.fct $SE_WS	experiment performance (only for experiment types UNC_MC with stopping rule, BAY_BC and OPT_SA)	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
<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

File / location	Generated in	Explanation
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 OPT_SA)	auxiliary files for experiment type OPT_SA
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. 11.1 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.

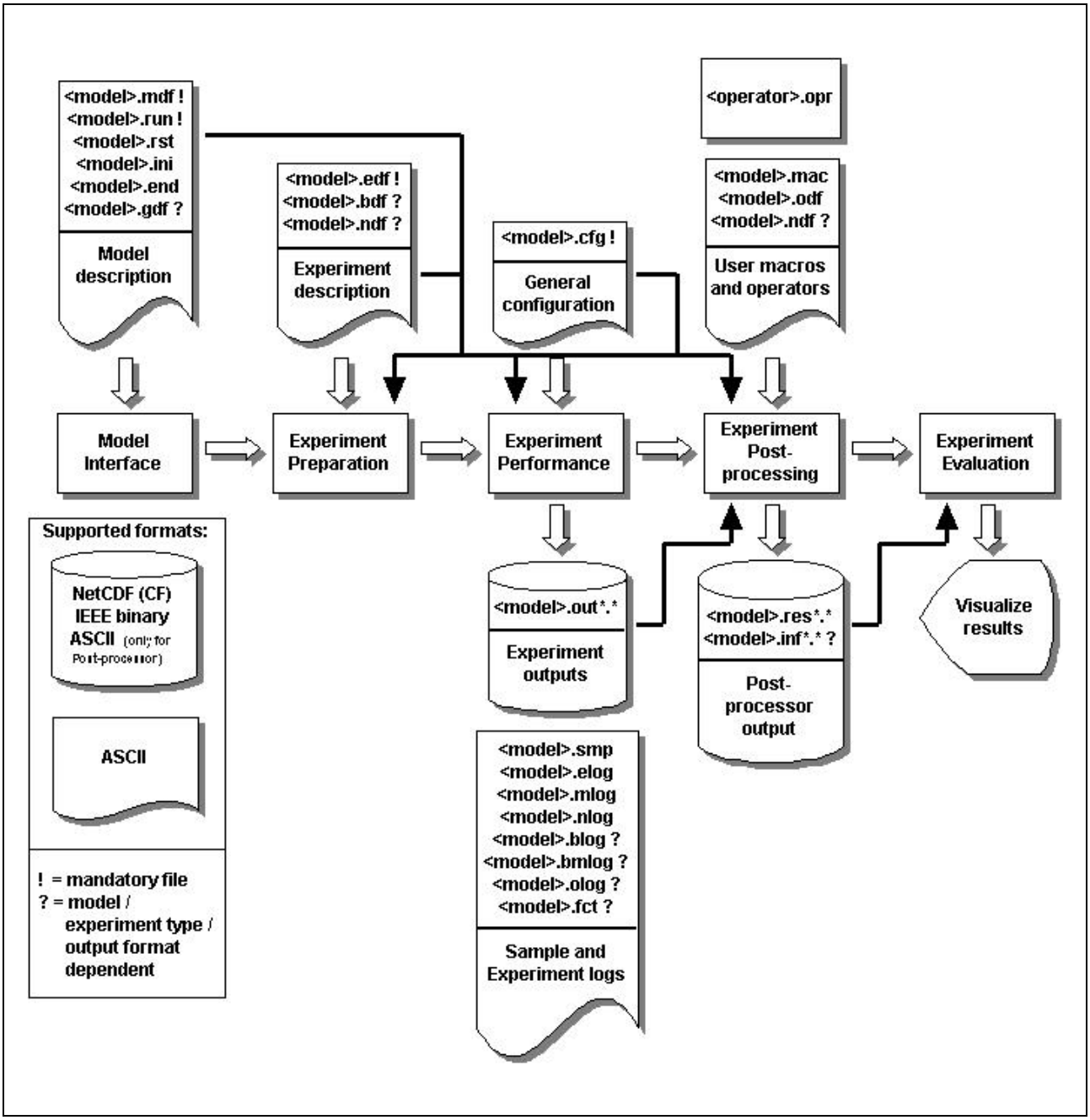


Fig. 11.1 *SimEnv user shell scripts and files*

11.6 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. 11.9 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. 11.9 *Built-in model output variables*

Built-in model output variable name	Dimensionality	Extents	Data type	Meaning
sim_time	0		float	elapsed user time in seconds when performing /usr/bin/time –p <model>.run

Tab. 11.10 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. 11.10 *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 (nominal) factor value for simenv_get_sh
simenv_hlp_*	auxiliary variables

Tab. 11.11 lists the built-in (pre-defined) coordinates that are used in experiment post-processing when additional dimensions are generated by an operator.

Tab. 11.11 *Built-in coordinates for experiment post-processing*

Built-in coordinate name	Generated by operator	Meaning	Definition (cf. Tab. 12.5)
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
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>	dfd	factor values	dependent on experiment description and operator arguments

11.7 Case Sensitivity

As stated in Tab. 11.12 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. 11.12 *Case sensitivity of SimEnv entities*

Where?	Entity	Case sen-sitiv-ity	Example
overall	• model name	yes	`simenv.chk World_f`
user-defined files (cf. Section 12.1)	• keyword • name • sub-keyword	no	`experiment  END_RUN  last`
	• information <value>	no	`experiment  end_run  LAST` `general  descr  This is ...`

model interface	• variable and factor name	no	`iok=simenv_put_f('ATMO',`
			`atmo,atmo)`
			`factor_name='P1'`
			`factor_value=1.`
			`. $SE_HOME/bin/simenv_get_sh`

experiment post-processing	• optional result description and unit	yes	`Energy [kW] = my_opr(atmo)`
	• character arguments of user-defined operators	yes	`char_test('arg11' , 'Arg21' ,atmo)`
	• variable and factor name • operator name • number • macro name • macro identifier _m	no	`3e-6*exp(atmo) + 3E-6*EXP(ATMO)`
	• character arguments of built-in operators with pre-defined values (cf. Tab. 15.10)	no	`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*])
		yes	`model  out_directory  MyDir`
			`factor  p1  sample  file MyF`
			`factor  p1  unit  kWh`

Where?	Entity	Case sen-sitiv-ity	Example
<model>.edf (for Mathematica models)	• <factor_name>	yes as in the model	`factor p1 sample list 1,2,3` `factor P1 sample list 3,4,5`
<model>.gdf (for GAMS models)	• GAMS model file name	yes	`model sub_m1 type sub` `model sub_M1 type sub`
<model>.ndf	• <attrib_name>	yes	`glob_attr delete attrib1` `glob_attr delete Attrib1`

11.8 Numerical Nodata Representation

For model output with the SimEnv model coupling interface functions and for experiment post-processor output default or user-defined data type specific nodata values are used to represent undefined (unwritten) model output and undefined post-processor output. For the latter check Section 8.8. Default nodata value representations are listed in Tab. 11.13. User-defined nodata values are specified in <model>.cfg as described in Section 11.1.

Tab. 11.13 *Data type related default nodata values*

SimEnv data type (cf. also Tab. 5.4)		Default nodata value
byte	int*1	127
short	int*2	32767
int	int*4	2147483647
float	real*4	3.40E+38
double	real*8	1.79D+308

All post-processor output is of type float / real*4. Model output in terms of model variables can be of any of the above types in Tab. 5.4 as defined in the file <model>.mdf, model output in terms of experiment factors is of type float / real*4.
An integer (byte / int*1, short / int*2, int / int*4) value is identified as nodata if the value exactly matches with the corresponding nodata value representation. In contrast, a real (float / real*4, double / real*8) value is identified as nodata if the value is in the neighbourhood of the corresponding nodata value representation. This neighbourhood is defined by its bounds $bound_{1/2}$ as follows:

For a float / real*4 data value <val_float>:

if abs(<val_float>) ≥ 1.e-34 then
  $bound_{1/2}$ = (1. ± 1.e-4) * <nodata_val_float>
else
  $bound_{1/2}$ = ±1.e-4
endif

For a double / real*8 data value <val_double>:

if abs(<val_double>) ≥ 2.d-307 then
  $bound_{1/2}$ = (1. ± 1.d-10) * <nodata_val_double>
else
  $bound_{1/2}$ = ±1.d-10
endif

Example:

For
<nodata_value*float> = -99999.
it is
$bound_1$ = -100009. and
$bound_2$ = -99989.

Within SimEnv post-processing all model and intermediate post-processor output values <val_float> with $bound_1 \leq$ <val_float> $\leq bound_2$ are identified as nodata values. That's why make sure that values of real model output variables are always outside the neighbourhood as defined above.

Important note:

The visualization system SimEnvVis (see Chapter 9) that is coupled to SimEnv expects that the nodata value assigned to the variable data type is outside the value range of this variable. The value range is bounded by the minimum and the maximum value of the variable. Normally, type dependent default nodata values are outside the value range. This may be violated for user-defined nodata values. For NetCDF model output each model output variable and for post-processor output each result comes with an attribute data_range that describes the value range by its boundaries (see explanations to Tab. 10.3). In particular, pay attention to a user-defined float / real*4 nodata type when post-processor output is stored in NetCDF format as post-processor NetCDF output files are visualized by SimEnvVis and all post-processor output is of type float / real*4.

11.9 Operating System Environment Variables

The following operating system environment variables are used by SimEnv.

Tab. 11.14 *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 services			
SE_BAY_BC_SETTING	setting number of the multiple setting case	--	provided by $SE_HOME/bin/simenv_bay_bc_sh within <model>.run (for experiment type BAY_BC) Value = <val_int> > 0
SE_GUI	identifier for GUI / non-GUI version	--	for all SimEnv services Value = [yes | <nil>]
SE_HN	generic hostname	--	for all SimEnv services Value = hostname without domain
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]

Additionally, make sure that in the shell the noclobber option is **not** set.
To perform SimEnv, make sure that paths to the directories of the programs as specified in Tab. 11.15 below are included in the environment variable PATH:

Tab. 11.15 *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
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. 11.14 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.

12 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.

12.1 General Structure of User-Defined Files

All user-defined files listed in Tab. 12.1 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 12.2 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. 11.12 on page 197.

Tab. 12.1 *User-defined files with general structure*

File	Contents	See description in Section	See description on page
<model>.cfg	general configuration file	11.1	181
<model>.mdf	model output description file	5.1	37
<model>.edf	experiment description file	6.1	69
<model>.odf	operator description file	8.5.4	162
<model>.mac	macro description file	8.6	163
<model>.gdf	GAMS description file	5.7.2	54
arbitrary file name	coordinate transformation file	12.2	205

The following restrictions hold for user-defined files:

Tab. 12.2 *Constraints in user-defined files*
***(*)**: with the exception for GAMS model source code file names*

Element	Constraints
line length	max. 360 characters
<name>	max. 64 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 120)
<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. 116 characters (for the resulting resolved directory string, directory can contain operating system environment variables)
	for sub-keyword = *unit*: max. 32 characters

Tab. 12.3 lists the reserved (forbidden) names and file names that cannot be declared in user-defined files.

Tab. 12.3 *Reserved names and file names in user-defined files*

Element	Reserved (forbidden) names
<name> (with the exception of GAMS model source code file names)	built-in model output variables according to Tab. 11.9
	built-in coordinates according to Tab. 11.11
	special keywords in <model>.edf for DFD: [default | file]
<file_name>	see Section 12.3

```
mac                 descr      This is a macro description file
mac                 descr      for the SimEnv User Guide

macro    np_atmo    descr      atmo outside polar reg., final time, level 1
macro    np_atmo    unit       atmo_unit
macro    np_atmo    define     atmo(c=84:-56,*,c=1,c=20)

macro    m1         define     avg(atmo_g(c=11:20))
...
```

Example 12.1 *Structure of a user-defined file*

12.2 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 11.1).

Tab. 12.4 *Elements of a coordinate transformation file (line type: o = optional)*

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	<val_float>	shifts all values of the original coordinate by the specified value <position_shift_val>
		values_shift	o	1	<val_int>	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. 12.5)

Keyword	Name	Sub-keyword	Line type	Max. line nmb.	Value	Explanation
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. 12.5)

To Tab. 12.4 the following additional rules and explanations apply:

- 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 44 and
assuming address_default = coordinate in <model>.cfg
Assume the experiment layout in Fig. 4.6 (c) on page 26 and
the corresponding experiment description file from Example 6.5 DFD_c on page 82.

Additionally, assume another experiment with a model named model and its
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 12.2 *Coordinate transformations by a transformation file*

12.3 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. 11.7 and Tab. 11.8
- 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. 12.5.

Tab. 12.5 *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$_2$>) <val_int>	description of an equidistant list of values with begin value <real_val$_1$> increment <real_val$_2$> number of values <val_int> <real_val$_2$> ≠ 0. <val_int> > 1

Value-list type	Syntax	Explanation
implicit with begin element number of values end element	equidist_ivl <real_val$_1$> (<val_int>) <real_val$_2$>	description of an equidistant list of values with begin value <real_val$_1$> number of values <val_int> end value <real_val$_2$> <val_int> > 1 <val_int> - 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 12.3 *Examples of value lists*

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

- for R models

Experiment preparation

- Experiment type stochastic analysis
- UNC_MC: 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

LLNL (2012): NetCDF Climate and Forecast (CF) Metadata Convention. Lawrence Livermore National Laboratory,
http://cf-pcmdi.llnl.gov/

NCO (2012): NetCDF operator Homepage
http://nco.sourceforge.net/

Pettitt, A.N. (1979): A non-parametric Approach to the Change-point Problem. Applied Statistics, 28, 126-135

Rabitz, H., Alis, Ö.F. (1999): General foundations of high-dimensional model representations. Journal of Mathematical Chemistry 95, 197-233

Saltelli, A., Chan, K., Scott, E.M. (eds.) (2000): Sensitivity Analysis. J. Wiley & Sons, Chichester

Saltelli, A. (2002): Making best use of model valuations to compute sensitivity indices. Computer Physics Communications 145, 280-297

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

SAS/STAT(R) 9.2 User's Guide.
http://support.sas.com/documentation/cdl/en/statug/63033/HTML/default/viewer.htm#statug_introbayes_sect008.htm
Accessed Feb 14, 2012.

Schulzweida, U. , Kornblueh, L., Quast, R. (2009): CDO - Climate Data Operators Version 1.4.1. Max-Planck-Institute for Meteorology, http://www.mpimet.mpg.de/fileadmin/software/cdo/

Sin, G., Gernaey, K.V. (2009): Improving the Morris method for sensitivity analysis by scaling the elementary effects. In: Jezowski, J., Thullie, J. (eds.) 19th European Symposium on Computer Aided Process Engineering – ESCAPE19. June 14-17, 2009, Cracow, Poland. Elsevier

Sobol' ,I.M. (1967): Distribution of points in a cube and approximate evaluation of integrals. U.S.S.R Comput. Maths. Math. Phys. 7, 86–112

Sobol', I.M. (1976): Uniformly distributed sequences with an additional uniform property. U.S.S.R Comput. Maths. Math. Phys. 16, 236–242

Sobol', I.M. (1993): Sensitivity analysis for non-linear mathematical models. Mathematical Modelling and Computational Experiment 1, 407-414

van Oijen, M: (2008) Bayesian calibration (BC) and Bayesian model comparison (BMC) of process-based models: Theory, implementation and guidelines. NERC, Centre for Ecology and Hydrology http://nora.nerc.ac.uk/6087/1/BC%26BMC_Guidance_2008-12-18_Final.pdf

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
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/	Intel ifort Version 10.0 http://www.intel.com/cd/software/products/asmo-na/eng/282048.htm or gfortran Version 4.2 GNU Fortran 95 compiler http://gcc.gnu.org/

Component	Specification	
	Unix	Linux
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/
MPI (only for performing experiments in compute cluster mode – see Section 7.3)	Version 1.0 http://www.mpi-forum.org	
For running the corresponding interfaced models		
GAMS	Distribution 20 http://www.gams.com	
Java	Version 1.4 http://www.java.com	
Matlab	Version 7.7 http://www.mathworks.com	
Mathematica	Version 4.1 http://wwwwolfram.com	
For the visualization framework SimEnvVis		
OpenDX	Version 4.4.4 http://www.opendx.org	
Qt	Version 3.3.5 http://www.trolltech.com/products/qt	
Ferret	Version 4.4 http://ferret.wrc.noaa.gov/Ferret/	
OpenGL	Version 1.4 http://www.opengl.org/	

The version number of the software products in the above Tab. 15.2 represent those version, SimEnv was developed with. Higher versions should also be applicable.

For setting up SimEnv, gunzip, tar, configure, make, a 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. 11.1).

15.1.2 Technical Limitations

Tab. 15.3 *Current SimEnv technical limitations*
***(*)**: Sampled factor values defined with equidist_[end | nmb | ivl] (see Tab. 12.5) allocate only 6 storage places.*

Entity		Maximum entity value
Directory strings ($SE_HOME/bin, current workspace; in user-defined files and operators)		
resolved length (relative to absolute paths, environ. variables resolved)	[characters]	256
User-defined files entities (cf. also Section 12.1)		
length of a record in a user-defined file	[characters]	360
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]	64
values in a value list	[number]	5 000
length of a record of a referred ASCII data file	[characters]	1 024
<model>.odf: user-defined and composed operators	[number]	27
<model>.ndf: total length of a global attribute string	[characters]	256
length of all define strings for a macro or a composed operator	[characters]	512
Model interface and experiment preparation entities		
single model runs in an experiment	[number]	999 999
experiment factors in <n´model>.edf	[number]	100
model output variables in <model>.mdf	[number]	100
all but Java models: dimensionality of a model output variable		9
Java models: dimensionality of a model output variable		4
Coordinates in <model>.mdf	[number]	100
coordinate values and sampled factor values **(*)**	[number]	200 000
active slices during performance of a model	[number]	30
GAMS models: storage size of a (float / real*4) model output variable	[Mbytes]	64
Experiment post-processing entities		
length of the optional result description string	[characters]	128
length of the optional result unit string	[characters]	32
arguments of an operator	[number]	9
dimensionality of a result		9
length of a complete result string (with optional description and unit)	[characters]	1 024
all operands and operators of a result	[number]	200
length of a character argument of an operator	[characters]	124
length of a string for a constant	[characters]	20
constants of a result	[number]	30
user-defined and composed operators	[number]	35

Entity		Maximum entity value
allocatable main memory segments for computing a result	[number]	10
total allocatable main memory	[MBytes]	2 048
results res_<digit><digit> in a NetCDF post-processor output file	[number]	99
post-processor output files <model>.res_<digit><digit>.[nc | ieee | ascii]	[number]	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/lib/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. 11.7 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	
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	

model	Language / source code	Explanation
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_[GSA_EE | GSA_VB | DFD_[a | b | c | d] | LSA | UNC_MC | BAY_BC | OPT_SA]	experiment description files corresponding to the individual experiment types 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_[GSA_EE | GSA_VB | DFD_c | LSA | UNC_MC | BAY_BC | OPT_SA]	Experiment specific post-processor input file (complete experiment) for the corresponding description files world.edf_<exp_type>: simenv.res world_[f | c | cpp | py | ja | m | sh] [new | append | replace] < world.post_<exp_type>
world.post_[bas | adv]	Experiment-type unspecific post-processor input file for single run post-processing of a selected single run <simenv_run_int>: simenv.res world_[f | c | cpp | py | ja | m | sh] [new | append | replace] <simenv_run_int> < world.post_[bas | adv]
world.dat_[DFD_d | UNC_MC | adv]	data files used in world.edf_[DFD_d | UNC_MC] 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 7 dynamics of the model world depend on four model parameters 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 (nominal) 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 semi-automated model interface 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*64 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,len_res)
c  SimEnv operator results are always of type real*4
   real*4 res(len_res)
   integer len_res
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];
/* 576 = 9 dimensions * 64 characters */
   char co_name1[576],co_name2[576];
   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 *len_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*64,res_nmb*2
      parameter (NR_DIM_MAX=9)
      real*4, pointer, dimension(:) :: coord_values
      real*4, pointer, dimension(:) :: result_values
      integer*4 idim, iext(NR_DIM_MAX)
      character result_expr*1024, coord_name*64, prefix*32, header*400
      character result_name*192, result_desc*128, result_unit*32
      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')
      do irec=1,3
         read(1,'(a33,a400)', iostat=iostat) prefix,header
      enddo
      iostat=0
      do while (iostat.eq.0)
         read(1,'(a33,a192)', iostat=iostat) prefix,result_name
         if(iostat.ne.0) exit
         read(1,'(a33,a128)', iostat=iostat1) prefix,result_desc
         read(1,'(a33,a1024)',iostat=iostat1) prefix,result_expr
         read(1,'(a33,a32)',  iostat=iostat1) prefix,result_unit
         read(1,'(a33,9i8)', iostat=iostat1) prefix,(iext(i),i=1,NR_DIM_MAX)
         length_result=1
         do i=1,NR_DIM_MAX
            if(iext(i).eq.0) exit
            length_result=length_result*iext(i)
            read(1,'(a33,a64)',iostat=iostat1) prefix,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 ...
   ...
            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 ...
   ...
         deallocate(result_values)
      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 Experiment Post-Processor Built-In Operators

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 120**
arg1 + arg2	addition	
arg1 - arg2	subtraction	
arg1 * arg2	multiplication	
arg1 / arg2	division	
arg1 **arg2	exponentiation	
+ arg1	identity	
- arg1	negation	
(arg1)	parentheses	
Basic operators		**Tab. 8.4 on page 121**
abs(arg1)	absolute value	
dim(arg1,arg2)	positive difference	
exp(arg1)	exponential function	
int(arg1)	integer truncation value	
log(arg1)	natural logarithm	
log10(arg1)	decade logarithm	
mod(arg1,arg2)	remainder	
nint(arg1)	nearest integer value	
sign(arg1)	sign of value	
round(int_arg1,int_arg2)	round value arg2 to int_arg1 decimal places	
sqrt(arg1)	square root	
Trigonometric operators		**Tab. 8.4 on page 121**
sin(arg1)	sine	
cos(arg1)	cosine	
tan(arg1)	tangent	
cot(arg1)	cotangent	
asin(arg1)	arc sine	
acos(arg1)	arc cosine	
atan(arg1)	arc tangent	

Name	Meaning See
acot(arg1)	arc cotangent
sinh(arg1)	hyperbolic sine
cosh(arg1)	hyperbolic cosine
tanh(arg1)	hyperbolic tangent
coth(arg1)	hyperbolic cotangent
Advanced operators	**Tab. 8.8 on page 126**
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
distr_par(char_arg1,int_arg2)	get distribution parameter int_arg2 of factor 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 arg2 (set them undefined)
{un}mask_file(char_arg1,arg2)	mask elements of arg2 (set them undefined) according to information from file char_arg1
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
usage(char_arg1)	get usage of post-proc. operator char_arg1
Aggregation and moment operators for arguments	**Tab. 8.5 on page 123**
avg(arg1)	argument arithmetic mean of values
avgg(arg1)	argument geometric mean of values
avgh(arg1)	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(arg1)	argument maximum of values
maxprop(arg1)	index of the element where the maximum is reached the first time
min(arg1)	argument minimum of values

Name	Meaning See
minprop(arg1)	index of the element where the minimum is reached the first time
sum(arg1)	argument sum of values
var(arg1)	argument variance of values
Multiple aggregation and moment operators for arguments	**Tab. 8.6 on page 124**
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 124**
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
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 (GSA_EE)	**Tab. 8.9 on page 138**
morris(arg1)	get global sensitivity measures for arg1
Multi-run operators (GSA_VB)	**Tab. 8.10 on page 139**
effects_1st(arg1)	linear effects of all factors for arg1
effects_tot(arg1)	total effects of all factors for arg1
gsa_vb_run_mask(char_arg1)	select sub-sample to apply to subsequent UNC_MC operators
Multi-run operators (DFD)	**Tab. 8.11 on page 142**
dfd(char_arg1,arg2)	general purpose operator for navigating and aggregating arg2 in the factor space
Multi-run operators (LSA)	**Tab. 8.14 on page 149**
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

Name	Meaning	See
sym_abs(char_arg1,arg2)	absolute symmetry measure	
sym_rel(char_arg1,arg2)	relative symmetry measure	
Multi-run operators (UNC_MC) also applicable for all other experiment types		**Tab. 8.13 on page 145**
avg_e(arg1)	run ensemble mean	
avgg_e(arg1)	run ensemble geometric mean	
avgh_e(arg1)	run ensemble harmonic mean	
avgw_e(arg1,arg2)	run ensemble weighted mean	
cnf_e(real_arg1,arg2)	positive distance of confidence line from run ensemble mean avg_e(arg2)	
cor_e (arg1,arg2)	run ensemble correlation coefficient between arg1 and arg2	
count_e(char_arg1,arg2)	run ensemble count number of values	
cov_e (arg1,arg2)	run ensemble covariance between arg1 and arg2	
ens(arg1)	whole run ensemble	
hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	heuristic probability density function	
krt_e (arg1)	run ensemble kurtosis (4^{th} moment)	
max_e(arg1)	run ensemble maximum	
maxprop_e(arg1)	run number where the maximum is reached the first time	
med_e (arg1)	run ensemble median	
min_e(arg1)	run ensemble minimum	
minprop_e(arg1)	run number where the minimum is reached the first time	
qnt_e (real_arg1,arg2)	run ensemble quantile of arg2	
reg_e (arg1,arg2)	run ensemble linear regression coefficient to forecast arg2 from arg1	
rng_e (arg1)	run ensemble range = max_e(arg1) - min_e(arg1)	
skw_e (arg1)	run ensemble skewness (3^{rd} moment)	
stat_full(real_arg1,real_arg2, real_arg3,real_arg4,arg5)	run ensemble full basic statistical measures	
stat_red(real_arg1,real_arg2, arg3)	run ensemble reduced basic statistical measures	
sum_e(arg1)	run ensemble sum	
var_e(arg1)	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)*
UNC_MC operators are also applicable for all other experiment types

Name	Meaning	Type	See	On page
arg1 + arg2	addition	elemental	Tab. 8.3	120
arg1 - arg2	subtraction	elemental	Tab. 8.3	120
arg1 * arg2	multiplication	elemental	Tab. 8.3	120
arg1 / arg2	division	elemental	Tab. 8.3	120
arg1 **arg2	exponentiation	elemental	Tab. 8.3	120
+ arg1	identity	elemental	Tab. 8.3	120
- arg1	negation	elemental	Tab. 8.3	120
(arg1)	parentheses	elemental	Tab. 8.3	120
abs(arg1)	absolute value	basic	Tab. 8.4	121
acos(arg1)	arc cosine	trigonom.	Tab. 8.4	121
acot(arg1)	arc cotangent	trigonom.	Tab. 8.4	121
asin(arg1)	arc sine	trigonom.	Tab. 8.4	121
atan(arg1)	arc tangent	trigonom.	Tab. 8.4	121
avg(arg1)	argument arithmetic mean of values	aggr./mom.	Tab. 8.5	123
avg_e(arg1)	run ensemble mean	UNC_MC	Tab. 8.13	145
avg_l(char_arg1,arg2)	dimension related argument arithmetic means of values of arg2	aggr./mom.	Tab. 8.7	124
avgg(arg1)	argument geometric mean of values	aggr./mom.	Tab. 8.5	123
avgg_e(arg1)	run ensemble geometric mean	UNC_MC	Tab. 8.13	145
avgg_l(char_arg1,arg2)	dimension related argument geometric means of values of arg2	aggr./mom.	Tab. 8.7	124
avgh(arg1)	argument harmonic mean of values	aggr./mom.	Tab. 8.5	123
avgh_e(arg1)	run ensemble harmonic mean	UNC_MC		
avgh_l(char_arg1,arg2)	dimension related argument harmonic means of values of arg2	aggr./mom.	Tab. 8.7	124
avgw(arg1,arg2)	argument weighted mean of values	aggr./mom.	Tab. 8.5	123
avgw_e(arg1,arg2)	run ensemble weighted mean	UNC_MC	Tab. 8.13	145
avgw_l(char_arg1,arg2, arg3)	dimension related argument weighted means of values of arg3	aggr./mom.	Tab. 8.7	124
classify(int_arg1,real_arg2, real_arg3,arg4)	classification of arg4 into int_arg1 classes	advanced	Tab. 8.8	126
clip(char_arg1,arg2)	clip arg2 according to char_arg1	advanced	Tab. 8.8	126
cnf_e(real_arg1,arg2)	positive distance of confidence line from run ensemble mean avg_e(arg2)	UNC_MC	Tab. 8.13	

Name	Meaning	Type	See	On page
cor_e (arg1,arg2)	run ensemble correlation coefficient between arg1 and arg2	UNC_MC	Tab. 8.13	145
cos(arg1)	cosine	trigonom.	Tab. 8.4	121
cosh(arg1)	hyperbolic cosine	trigonom.	Tab. 8.4	121
cot(arg1)	cotangent	trigonom.	Tab. 8.4	121
coth(arg1)	hyperbolic cotangent	trigonom.	Tab. 8.4	121
count(char_arg1,arg2)	count number of values	aggr./mom.	Tab. 8.5	123
count_e(char_arg1,arg2)	run ensemble count	UNC_MC	Tab. 8.13	145
count_l(char_arg1, char_arg2,arg3)	dimension related count numbers of values of arg3	aggr./mom.	Tab. 8.7	124
cov_e (arg1,arg2)	run ensemble covariance between arg1 and arg2	UNC_MC	Tab. 8.13	145
cumul(char_arg1,arg2)	cumulates arg2 according to char_arg1	advanced	Tab. 8.8	126
dfd(char_arg1,arg2)	general purpose operator for navigating and aggregating of arg2 in the factor space	DFD	Tab. 8.11	142
dim(arg1,arg2)	positive difference	basic	Tab. 8.4	121
distr(char_arg1,int_arg2)	get distribution parameter int_arg2 of factor char_arg1	basic	Tab. 8.4	121
effects_1st(arg1)	linear effects of all factors for arg1		Tab. 8.10	139
effects_tot(arg1)	total effects of all factors for arg1		Tab. 8.10	139
ens(arg1)	whole run ensemble	UNC_MC	Tab. 8.13	145
exp(arg1)	exponential function	basic	Tab. 8.4	121
flip(char_arg1,arg2)	flip arg2 according to char_arg1	advanced	Tab. 8.8	126
get_data(char_arg1, char_arg2,char_arg3,arg4)	get data from an external file	advanced	Tab. 8.8	126
get_experiment(char_arg1, char_arg2,char_arg3,arg4)	include an other experiment	advanced	Tab. 8.8	126
get_table_fct(char_arg1, arg2)	table function with linear interpolation of table char_arg1 for position arg2	advanced	Tab. 8.8	126
gsa_vb_run_mask(char_arg1)	select sub-sample to apply to subsequent UNC_MC operators	GSA_VB	Tab. 8.10	139
hgr(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	argument histogram of values	aggr./mom.	Tab. 8.5	123
hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)	run ensemble heuristic probability density function	UNC_MC	Tab. 8.13	145
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	124
if(char_arg1,arg2,arg3,arg4)	general purpose conditional if-construct	advanced	Tab. 8.8	126
int(arg1)	integer truncation value	basic	Tab. 8.4	121
krt_e (arg1)	run ensemble kurtosis (4^{th} moment)	UNC_MC	Tab. 8.13	145
lin_abs(char_arg1,arg2)	absolute linearity measure	loc. sens.	Tab. 8.14	149
lin_rel(char_arg1,arg2)	relative linearity measure	loc. sens.	Tab. 8.14	149
log(arg1)	natural logarithm	basic	Tab. 8.4	121
log10(arg1)	decade logarithm	basic	Tab. 8.4	121

Name	Meaning	Type	See	On page
mask(char_arg1,arg2,arg3)	mask elements of argument arg2 (set them undefined)	advanced	Tab. 8.8	126
{un}mask_file(char_arg1, arg2)	mask elements of argument arg2 (set them undefined) according to information in file char_arg1	advanced	Tab. 8.8	126
matmul(arg1,arg2)	matrix multiplication	advanced	Tab. 8.8	126
max(arg1)	argument maximum of values	aggr./mom.	Tab. 8.5	123
max_e(arg1)	run ensemble maximum	UNC_MC		
max_l(char_arg1,arg2)	dimension related argument maxima of values of arg2	aggr./mom.	Tab. 8.7	124
max_n(arg1 ,..., argn)	maximum per element	aggr./mom.	Tab. 8.5	124
maxprop(arg1)	index of the element where the maximum is reached the first time	aggr./mom.	Tab. 8.5	123
maxprop_e(arg1)	run number where the maximum is reached the first time	UNC_MC	Tab. 8.13	145
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	124
maxprop_n(arg1 ,..., argn)	argument position (1 ,..., n) where the maximum is reached the first time	aggr./mom.	Tab. 8.5	124
med_e (arg1)	run ensemble median	UNC_MC	Tab. 8.13	145
min(arg1)	argument minimum of values	aggr./mom.	Tab. 8.5	123
min_e(arg1)	run ensemble minimum	UNC_MC		
min_l(char_arg1,arg2)	dimension related argument minima of values of arg2	aggr./mom.	Tab. 8.7	124
min_n(arg1 ,..., argn)	minimum per element	aggr./mom.	Tab. 8.5	124
minprop(arg1)	index of the element where the minimum is reached the first time	aggr./mom.	Tab. 8.5	123
minprop_e(arg1)	run number where the minimum is reached the first time	UNC_MC	Tab. 8.13	145
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	124
minprop_n(arg1 ,..., argn)	argument position (1 ,..., n) where the minimum is reached the first time	aggr./mom.	Tab. 8.5	124
mod(arg1,arg2)	remainder	basic	Tab. 8.4	121
morris(arg1)	get global sensitivity measures for arg1	glob. sens.	Tab. 8.9	138
move_avg(char_arg1, char_arg2,int_arg3,arg4)	moving average of running length int_arg3 for arg4	advanced	Tab. 8.8	126
nint(arg1)	nearest integer value	basic	Tab. 8.4	121
qnt_e (real_arg1,arg2)	run ensemble quantile of arg2	UNC_MC	Tab. 8.13	145
rank(char_arg1,arg2)	rank of arg2 according to char_arg1	advanced	Tab. 8.8	126
reg_e (arg1,arg2)	run ensemble linear regression coefficient to forecast arg2 from arg1	UNC_MC	Tab. 8.13	145
regrid(char_arg1,arg2)	assign new coordinates to arg2	advanced	Tab. 8.8	126
rng_e (arg1)	run ensemble range = max_e(arg1) - min_e(arg1)	UNC_MC	Tab. 8.13	145

Name	Meaning	Type	See	On page
round(int_arg1,arg2)	round value arg2 to int_arg1 decimal places	basic	Tab. 8.4	121
run(char_arg1,arg2)	values of arg2 for a single run selected by char_arg1	advanced	Tab. 8.8	126
run_info(char_arg1)	current run number and/or number of single runs of the current experiment	advanced	Tab. 8.8	126
sens_abs(char_arg1,arg2)	absolute sensitivity measure	loc. sens.	Tab. 8.14	149
sens_rel(char_arg1,arg2)	relative sensitivity measure	loc. sens.	Tab. 8.14	149
sign(arg1)	sign of value	basic	Tab. 8.4	121
sin(arg1)	sine	basic	Tab. 8.4	121
sinh(arg1)	hyperbolic sine	trigonom.	Tab. 8.4	121
skw_e (arg1)	run ensemble skewness (3rd moment)	UNC_MC	Tab. 8.13	145
sqrt(arg1)	square root	trigonom.	Tab. 8.4	121
stat_full(real_arg1, real_arg2,real_arg3, real_arg4,arg5)	full basic statistical measures	UNC_MC	Tab. 8.13	145
stat_red(real_arg1, real_arg2,arg3)	reduced basic statistical measures	UNC_MC	Tab. 8.13	145
sum(arg1)	argument sum of values	aggr./mom.	Tab. 8.5	123
sum_e(arg1)	run ensemble sum	UNC_MC		
sum_l(char_arg1,arg2)	dimension related argument sums of values of arg2	aggr./mom.	Tab. 8.7	124
sym_abs(char_arg1,arg2)	absolute symmetry measure	loc. sens.	Tab. 8.14	149
sym_rel(char_arg1,arg2)	relative symmetry measure	loc. sens.	Tab. 8.14	149
tan(arg1)	tangent	trigonom.	Tab. 8.4	121
tanh(arg1)	hyperbolic tangent	trigonom.	Tab. 8.4	121
transpose(char_arg1,arg2)	transpose arg2 according to char_arg1	advanced	Tab. 8.8	126
undef( )	undefined element	advanced	Tab. 8.8	126
usage(char_arg1)	get usage of post-proc. operator char_arg1	advanced	Tab. 8.8	126
var(arg1)	argument variance of values	aggr./mom.	Tab. 8.5	123
var_e(arg1)	run ensemble variance	UNC_MC	Tab. 8.13	145
var_l(char_arg1,arg2)	dimension related argument variances of values of arg2	aggr./mom.	Tab. 8.7	124

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	Argu-ment number	Argument value (pre-defined values are case-insensitive)	Remark
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	(**)
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	(**)
dfd	1	(not pre-defined, case insensitive)	(*)
distr_par	2	factor name, case-insensitive	
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	[‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘!=’]	
{un}mask_file	1	{<directory>/}<file_name>	
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	(**)

Operator	Argu-ment number	Argument value (pre-defined values are case-insensitive)	Remark
move_avg	1	sequence of digits 0 to 9	(**)
move_avg	2	['lin' | 'exp']	
rank	1	[,tie_plain' | ,tie_min' | 'tie_avg']	
regrid	1	{<directory>/}<file_name>	
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]
distr_par	2	int_arg	≥ 2
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]
qnt	1	real_arg	0. ≤ arg1 ≤ 100.
round	1	int_arg	1 ≤ arg1 ≤ 5
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	

15.5.5 Experiment Post-Processor Built-In Unit Invariant Operators

Tab. 15.12 lists those built-in experiment post-processor operators that have only one non-constant / non-character argument and are invariant with respect to the unit of the operand. For more information check Section 8.1.1.

Tab. 15.12 *Experiment post-processor unit invariant built-in operators (operators marked by **(*)** have more than one non-constant / non-character argument)*

Operator		Operator
monadic -		lin_abs
abs		log
acos		log10
acot		mask_file
asin		max
atan		max_e
avg		max_l
avg_e		max_n (*)
avg_l		med_e
avgg_e		min
avgh_e		min_e
avgw (*)		min_l
avgw_e (*)		min_n (*)
avgw_l (*)		nint
bay_bc_run_mask		qnt_e
clip		rng_e
cnf_e		round
cos		sens_abs
cosh		sin
cot		sinh
coth		sqrt
cumul		sum
dfd		sum_e
distr_par		sum_l
exp		sym_abs
flip		tan
gsa_vb_run_mask		tanh
int		unmask_file

15.6 Additionally Used Symbols for the Model and Operator Interface

Tab. 15.13 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. 15.13 *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. 15.14 lists those symbols (subroutine, function and common block names) that are linked in addition to the SimEnv operator interface functions in Tab. 8.19 and Tab. 8.20 from the object library $SE_HOME/lib/libsimenv.a to a user-defined experiment post-processing operator.

Tab. 15.14 *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.

BAY_BC (Bayesian calibration): → Experiment type to reduce uncertainty about → factor values by deriving a representative → sample from factor prior distributions while having measurement values from the system available for → model – measurement comparison.

Column-major order model: A rule how to store the elements of a multi-dimensional data array to a 1-dimensional linear vector representation and *vice versa*. A multi-dimensional data array array(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 on a 1-dimensional vector vector(1:ext_1 * ext_2 * ... * ext_{dim-1} * ext_{dim}) of extent 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) = array(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. The column-major order model is used in Fortran and → Matlab. The opposite model where rows of a matrix are listed in sequence is called row-major order model and is used in C/C++.

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.

DFD (Deterministic Factorial Design): → 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.

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 SimEnv scripts but those marked with (*) in Tab. 11.5 at page 188 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 → GSA_EE, → DFD, → UNC_MC, → LSA, and → OPT_SA.

Expression: → result expression

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.

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

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.

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.

GSA_EE (Global sensitivity analysis – elementary effects method): → 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 based on a statistics from local effects on randomly selected trajectories in the factor space.

GSA_VB (Global sensitivity analysis – variance based): → Experiment type to determine how the variance of model output depends on the variability of the model → factors and how the output variance can be apportioned accordingly.

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. See also → PBS / Torque.

LSA (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, initial values or driving forces 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 the SimEnv 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

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.

OPT_SA (Optimization – Simulated Annealing): → Experiment type to minimize a cost function (objective function) over a bounded → factor space. In SimEnv a simulated annealing strategy (cf. Section 4.7 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

PBS / Torque: Portable Batch System (or simply PBS) is the name of computer software that performs job scheduling, and provides functions for building, submitting, and processing them. Torque is a fork of OpenPBS, the open source version of PBS. See also → Load Leveler.

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.

Row-major order model: A rule how to store the elements of a multi-dimensional data array to a 1-dimensional linear vector representation and *vice versa*. → column-major order model

Sample: A set of numerical → factor values created during experiment preparation or performance.

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 an → experiment with a → model

UNC_MC (Uncertainty analysis – Monte Carlo method): → 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 experiment 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.

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, and → macro 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 an array.

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 is started from and the SimEnv user-defined files such as <model>.mdf, <model>.edf, <model>.run, <model>.cfg are located in.