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


by

Michael Flechsig  Potsdam Institute for Climate Impact Research
Department Data & Computation, flechsig@pik-potsdam.de

Uwe Böhm  Potsdam Institute for Climate Impact Research
Climate System Department, boehm@pik-potsdam.de

Thomas Nocke  University of Rostock
Institute of Computer Graphics, nocke@informatik.uni-rostock.de

Claus Rachimow  Potsdam Institute for Climate Impact Research
Department Data & Computation, 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 2600
WWW  http://www.pik-potsdam.de

University of Rostock  Institute of Computer Graphics
Albert-Einstein-Str. 21
18059 Rostock, Germany
Phone  ++49 – 381 – 498 7481
Fax  ++49 – 381 – 498 7482
WWW  http://wwwicg.informatik.uni-rostock.de
Contents

EXECUTIVE SUMMARY ........................................................................................................... 1

1 ABOUT THIS DOCUMENT ........................................................................................................ 3
1.1 Document Conventions ........................................................................................................ 3
1.2 Example Layout .................................................................................................................. 4

2 GETTING STARTED ............................................................................................................... 5

3 VERSION 1.22 .......................................................................................................................... 7
3.1 What is New? ....................................................................................................................... 7
3.2 Limitations / Problems and Their Workarounds ................................................................. 8
3.3 Known Bugs and Their Workarounds ................................................................................. 9

4 EXPERIMENT TYPES ......................................................................................................... 11
4.1 General Approach ............................................................................................................. 11
4.2 Behavioural Analysis ........................................................................................................ 13
4.3 Monte Carlo Analysis ........................................................................................................ 14
4.4 Local Sensitivity Analysis ................................................................................................ 16
4.5 Optimization .................................................................................................................... 18

5 MODEL INTERFACE ........................................................................................................... 21
5.1 General Approach ............................................................................................................. 21
5.2 Grid and Coordinate Assignments to Variables ............................................................... 24
5.3 Model Output Description File <model>.mdf .................................................................. 25
5.4 Model Interface for Fortran and C/C++ Models .............................................................. 27
5.5 Model Interface for Python Models .................................................................................. 30
5.5.1 Standard Shell Scripts for Python Models ................................................................. 31
5.6 Model Interface at Shell Script Level ................................................................................ 31
5.7 Model Interface for GAMS Models .................................................................................. 33
5.7.1 Standard Shell Scripts for GAMS Models ................................................................. 34
5.7.2 GAMS Description File <model>.gdf ...................................................................... 36
5.7.3 Files Created during GAMS Model Performance ....................................................... 38
5.8 Semi-Automated Model Interface .................................................................................... 39
5.9 Supported Model Structures ............................................................................................ 40
5.10 Using Interfaced Models Outside SimEnv ....................................................................... 41

6 EXPERIMENT PREPARATION ......................................................................................... 43
6.1 General Approach - Experiment Description File <model>.edf .................................... 43
6.2 Behavioural Analysis ........................................................................................................ 44
6.2.1 The Combination .......................................................................................................... 45
6.2.2 Example ......................................................................................................................... 46
6.3 Monte Carlo Analysis ....................................................................................................... 47
6.3.1 Distribution Functions and their Parameters ............................................................... 47
6.3.2 Example ......................................................................................................................... 48
6.4 Local Sensitivity Analysis ............................................................................................... 49
6.4.1 Sensitivity Functions and Run Sequence ................................................................. 49
6.4.2 Example ......................................................................................................................... 50
6.5 Optimization .................................................................................................................... 51
6.5.1 Special Features in Optimization ................................................................................. 51
6.5.2 Example ......................................................................................................................... 52

7 EXPERIMENT PERFORMANCE .................................................................................... 53
7.1 General Approach ............................................................................................................. 53
7.2 Model Wrap Shell Script <model>.run, Experiment-Specific Preparation and Wrap-Up Shell Scripts .......................... 54
7.3 Experiment Parallelization .............................................................................................. 55
7.4 Experiment Restart ......................................................................................................... 56
7.5 Experiment Partial Performance ..................................................................................... 58
7.6 Experiment Related User Shell Scripts and Files .......................................................... 59
7.7 Saving Experiments ........................................................................................................ 61

8 EXPERIMENT POST-PROCESSING ........................................................................... 63
8.1 General Approach ........................................................................................................... 63
8.1.1 Post-Processor Results ............................................................................................... 63
8.1.2 Operands ....................................................................................................................... 64
8.1.3 Model Output Variables .............................................................................................. 65
8.1.4 Operators .................................................................................................................... 67
15.3 Example Implementation for the Experiment Post-Processor User-Defined Operator matmul_[ f | c ]..... 155
15.3.1 Fortran Implementation ............................................................ 155
15.3.2 C Implementation ........................................................................ 158
15.4 Example for an Experiment Post-Processor Result Import Interface ........................................... 161
15.5 List of Experiment Post-Processor Built-In Operators and Operator Arguments .......................... 162
15.5.1 Experiment Post-Processor Built-In Operators (in Thematic Order) ........................................ 162
15.5.2 Experiment Post-Processor Built-In Operators (in Alphabetic Order) ...................................... 165
15.5.3 Character Arguments of Experiment Post-Processor Built-In Operators ............................... 168
15.5.4 Constant Arguments of Experiment Post-Processor Built-In Operators ................................. 169
15.6 Additionally Used Symbols for the Model and Operator Interface .............................................. 170
15.7 Glossary .......................................................................................... 171

Tables

<table>
<thead>
<tr>
<th>Tab.</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tab. 1.1</td>
<td>Document conventions</td>
<td>3</td>
</tr>
<tr>
<td>Tab. 1.2</td>
<td>Main placeholders in this document</td>
<td>3</td>
</tr>
<tr>
<td>Tab. 3.1</td>
<td>SimEnv changes in Version 1.22</td>
<td>7</td>
</tr>
<tr>
<td>Tab. 3.2</td>
<td>User actions to upgrade to Version 1.22</td>
<td>8</td>
</tr>
<tr>
<td>Tab. 3.3</td>
<td>Limitations / problems and their workarounds</td>
<td>9</td>
</tr>
<tr>
<td>Tab. 3.4</td>
<td>Known bugs and their workarounds</td>
<td>9</td>
</tr>
<tr>
<td>Tab. 4.1</td>
<td>Statistical measures</td>
<td>15</td>
</tr>
<tr>
<td>Tab. 4.2</td>
<td>Probability density functions</td>
<td>16</td>
</tr>
<tr>
<td>Tab. 4.3</td>
<td>Local sensitivity, linearity, and symmetry measures</td>
<td>17</td>
</tr>
<tr>
<td>Tab. 5.1</td>
<td>Generic SimEnv interface functions</td>
<td>21</td>
</tr>
<tr>
<td>Tab. 5.2</td>
<td>Language suffixes for SimEnv interface functions</td>
<td>22</td>
</tr>
<tr>
<td>Tab. 5.3</td>
<td>Elements of a model output description file &lt;model&gt;.mdf</td>
<td>25</td>
</tr>
<tr>
<td>Tab. 5.4</td>
<td>SimEnv data types</td>
<td>26</td>
</tr>
<tr>
<td>Tab. 5.5</td>
<td>Model interface functions for Fortran and C/C++ models</td>
<td>29</td>
</tr>
<tr>
<td>Tab. 5.6</td>
<td>Model interface functions for Python models</td>
<td>30</td>
</tr>
<tr>
<td>Tab. 5.7</td>
<td>Model interface functions at shell script level</td>
<td>32</td>
</tr>
<tr>
<td>Tab. 5.8</td>
<td>Elements of a GAMS description file &lt;model&gt;.gdf</td>
<td>37</td>
</tr>
<tr>
<td>Tab. 6.1</td>
<td>Elements of an experiment description file &lt;model&gt;.edf</td>
<td>43</td>
</tr>
<tr>
<td>Tab. 6.2</td>
<td>Adjustment types in experiment preparation</td>
<td>44</td>
</tr>
<tr>
<td>Tab. 6.3</td>
<td>Experiment specific elements of an edf-file for behavioural analysis</td>
<td>44</td>
</tr>
<tr>
<td>Tab. 6.4</td>
<td>Experiment specific elements of an edf-file for Monte Carlo analysis</td>
<td>47</td>
</tr>
<tr>
<td>Tab. 6.5</td>
<td>Probability density functions and their parameters</td>
<td>47</td>
</tr>
<tr>
<td>Tab. 6.6</td>
<td>Experiment specific elements of an edf-file for local sensitivity analysis</td>
<td>49</td>
</tr>
<tr>
<td>Tab. 6.7</td>
<td>Experiment specific elements of an edf-file for an optimization experiment</td>
<td>51</td>
</tr>
<tr>
<td>Tab. 7.1</td>
<td>Experiment related user shell scripts and files</td>
<td>59</td>
</tr>
<tr>
<td>Tab. 8.1</td>
<td>Classified argument restriction(s) / result description</td>
<td>68</td>
</tr>
<tr>
<td>Tab. 8.2</td>
<td>Built-in generic standard aggregation / moment operators</td>
<td>70</td>
</tr>
<tr>
<td>Tab. 8.3</td>
<td>Built-in elemental operators</td>
<td>70</td>
</tr>
<tr>
<td>Tab. 8.4</td>
<td>Built-in basic and trigonometric operators</td>
<td>71</td>
</tr>
<tr>
<td>Tab. 8.5</td>
<td>Built-in standard aggregation / moment operators without suffix</td>
<td>73</td>
</tr>
<tr>
<td>Tab. 8.6</td>
<td>Built-in standard aggregation / moment operators with suffix _n</td>
<td>73</td>
</tr>
<tr>
<td>Tab. 8.7</td>
<td>Built-in standard aggregation / moment operators with suffix _l</td>
<td>74</td>
</tr>
<tr>
<td>Tab. 8.8</td>
<td>Built-in advanced operators</td>
<td>77</td>
</tr>
<tr>
<td>Tab. 8.9</td>
<td>Multi-run standard aggregation / moment operators</td>
<td>83</td>
</tr>
<tr>
<td>Tab. 8.10</td>
<td>Experiment specific operator for behavioural analysis</td>
<td>84</td>
</tr>
<tr>
<td>Tab. 8.11</td>
<td>Syntax of the filter argument 1 for operator behav</td>
<td>85</td>
</tr>
<tr>
<td>Tab. 8.12</td>
<td>Experiment specific operators for Monte Carlo analysis</td>
<td>88</td>
</tr>
<tr>
<td>Tab. 8.13</td>
<td>Experiment specific operators for local sensitivity analysis</td>
<td>90</td>
</tr>
<tr>
<td>Tab. 8.14</td>
<td>Syntax of the filter argument 1 for local sensitivity operators</td>
<td>90</td>
</tr>
<tr>
<td>Tab. 8.15</td>
<td>Operator interface functions for the declarative and computational part</td>
<td>93</td>
</tr>
<tr>
<td>Tab. 8.16</td>
<td>Operator interface functions to get and put structural information</td>
<td>96</td>
</tr>
<tr>
<td>Tab. 8.17</td>
<td>Operator interface functions to get / check / put arguments and results</td>
<td>97</td>
</tr>
<tr>
<td>Tab. 8.18</td>
<td>Elements of an operator description file &lt;model&gt;.odf</td>
<td>100</td>
</tr>
<tr>
<td>Tab. 8.19</td>
<td>Elements of a macro description file &lt;model&gt;.mac</td>
<td>101</td>
</tr>
</tbody>
</table>
### Figures

<table>
<thead>
<tr>
<th>Fig. 0.1</th>
<th>SimEnv system design</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fig. 4.1</td>
<td>Target space</td>
</tr>
<tr>
<td>Fig. 4.2</td>
<td>Sample for a behavioural analysis</td>
</tr>
<tr>
<td>Fig. 4.3</td>
<td>Behavioural analysis: Scanning multi-dimensional target spaces</td>
</tr>
<tr>
<td>Fig. 4.4</td>
<td>Sample for a Monte Carlo analysis</td>
</tr>
<tr>
<td>Fig. 4.5</td>
<td>Sample for a local sensitivity analysis</td>
</tr>
<tr>
<td>Fig. 4.6</td>
<td>Part of a sample for an optimization experiment, generated during the experiment</td>
</tr>
<tr>
<td>Fig. 5.1</td>
<td>Conceptual scheme of the model interface for C/C++/Fortran/Python</td>
</tr>
<tr>
<td>Fig. 5.2</td>
<td>Grid types</td>
</tr>
<tr>
<td>Fig. 5.3</td>
<td>Model output variable definition: Grid assignment</td>
</tr>
<tr>
<td>Fig. 6.1</td>
<td>Monte Carlo analysis: Latin hypercube sampling</td>
</tr>
<tr>
<td>Fig. 7.1</td>
<td>Flowcharts for performing simenv.run and simenv.rst</td>
</tr>
<tr>
<td>Fig. 10.1</td>
<td>SimEnv user shell scripts and files</td>
</tr>
</tbody>
</table>

### Tables

| Tab. 10.1 | Elements of a general configuration file <model>.cfg |
| Tab. 10.2 | Default values for the general configuration file |
| Tab. 10.3 | SimEnv services |
| Tab. 10.4 | SimEnv and user include files and link scripts |
| Tab. 10.5 | Contents of $SE_HOME/simenv_mod_[f | c|].inc |
| Tab. 10.6 | User files and shell scripts to perform any SimEnv service |
| Tab. 10.7 | Files generated during performance of SimEnv services |
| Tab. 10.8 | Built-in model output variables |
| Tab. 10.9 | Built-in coordinates |
| Tab. 10.10 | Built-in shell script variables in <model>.run |
| Tab. 10.11 | Case sensitivity of SimEnv entities |
| Tab. 10.12 | Data type related nodata values |
| Tab. 10.13 | Environment variables |
| Tab. 11.1 | User-defined files with general structure |
| Tab. 11.2 | Constraints in user-defined files |
| Tab. 11.3 | Reserved names and file names in user-defined files |
| Tab. 11.4 | Line types in user-defined files |
| Tab. 11.5 | Elements of a coordinate transformation file |
| Tab. 11.6 | Syntax rules for value lists |
| Tab. 12.1 | NetCDF data types |
| Tab. 12.2 | Additional global NetCDF attributes |
| Tab. 12.3 | Variable NetCDF attributes |
| Tab. 12.4 | Variable NetCDF attributes for visualization |
| Tab. 15.1 | SimEnv installation under Unix |
| Tab. 15.2 | System requirements |
| Tab. 15.3 | Current SimEnv technical limitations |
| Tab. 15.4 | Implemented example models for the current version |
| Tab. 15.5 | Implemented model and operator related user files for the current version |
| Tab. 15.6 | Available user-defined operators |
| Tab. 15.7 | Targets of the generic model world |
| Tab. 15.8 | Experiment post-processor built-in operators (in thematic order) |
| Tab. 15.9 | Experiment post-processor built-in operators (in alphabetical order) |
| Tab. 15.10 | Character arguments of experiment post-processor built-in operators |
| Tab. 15.11 | Constant arguments of experiment post-processor built-in operators |
| Tab. 15.12 | Additionally used symbols for the model interface |
| Tab. 15.13 | Additionally used symbols for the operator interface |
### Examples

<table>
<thead>
<tr>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example 1.1</td>
<td>General example layout in the User Guide</td>
</tr>
<tr>
<td>Example 5.1</td>
<td>Model output description file &lt;model&gt;.mdf</td>
</tr>
<tr>
<td>Example 5.2</td>
<td>Addressing target names and values for the model interface at shell script level</td>
</tr>
<tr>
<td>Example 5.3</td>
<td>Model output description file for a GAMS model</td>
</tr>
<tr>
<td>Example 5.4</td>
<td>GAMS description file &lt;model&gt;.gdf</td>
</tr>
<tr>
<td>Example 5.5</td>
<td>GAMS description file for coupled GAMS models</td>
</tr>
<tr>
<td>Example 5.6</td>
<td>Shell script &lt;model&gt;.run for a parallel model</td>
</tr>
<tr>
<td>Example 6.1</td>
<td>Experiment description file &lt;model&gt;.edf for behavioural analysis</td>
</tr>
<tr>
<td>Example 6.2</td>
<td>Experiment description file &lt;model&gt;.edf for Monte Carlo analysis</td>
</tr>
<tr>
<td>Example 6.3</td>
<td>Experiment description file &lt;model&gt;.edf for local sensitivity analysis</td>
</tr>
<tr>
<td>Example 6.4</td>
<td>Experiment description file &lt;model&gt;.edf for an optimization experiment</td>
</tr>
<tr>
<td>Example 7.1</td>
<td>Shell script &lt;model&gt;.run to wrap the user model</td>
</tr>
<tr>
<td>Example 7.2</td>
<td>Shell script &lt;model&gt;.ini for user-model specific experiment preparation</td>
</tr>
<tr>
<td>Example 7.3</td>
<td>Shell script &lt;model&gt;.end for user-model specific experiment wrap-up</td>
</tr>
<tr>
<td>Example 7.4</td>
<td>Shell script &lt;model&gt;.rst to prepare model performance during experiment restart</td>
</tr>
<tr>
<td>Example 8.1</td>
<td>Addressing results in experiment post-processing</td>
</tr>
<tr>
<td>Example 8.2</td>
<td>Addressing model output variables in experiment post-processing</td>
</tr>
<tr>
<td>Example 8.3</td>
<td>Checking rules for coordinates</td>
</tr>
<tr>
<td>Example 8.4</td>
<td>Experiment post-processing operator get_data and coordinate transformation file</td>
</tr>
<tr>
<td>Example 8.5</td>
<td>Experiment post-processing with advanced operators</td>
</tr>
<tr>
<td>Example 8.6</td>
<td>Experiment post-processing operator behav for behavioural analysis</td>
</tr>
<tr>
<td>Example 8.7</td>
<td>Experiment post-processing operators for Monte Carlo analysis</td>
</tr>
<tr>
<td>Example 8.8</td>
<td>Experiment post-processing operators for local sensitivity analysis</td>
</tr>
<tr>
<td>Example 8.9</td>
<td>Composed operators</td>
</tr>
<tr>
<td>Example 8.10</td>
<td>Operator description file &lt;model&gt;.odf</td>
</tr>
<tr>
<td>Example 8.11</td>
<td>User-defined macro definition file &lt;model&gt;.mac</td>
</tr>
<tr>
<td>Example 8.12</td>
<td>Experiment post-processing with wildcard operands</td>
</tr>
<tr>
<td>Example 10.1</td>
<td>User-defined general configuration file &lt;model&gt;.cfg</td>
</tr>
<tr>
<td>Example 11.1</td>
<td>Structure of a user-defined file</td>
</tr>
<tr>
<td>Example 11.2</td>
<td>Coordinate transformations by a transformation file</td>
</tr>
<tr>
<td>Example 11.3</td>
<td>Examples of value lists</td>
</tr>
<tr>
<td>Example 12.1</td>
<td>IEEE compliant model output data structure</td>
</tr>
<tr>
<td>Example 15.1</td>
<td>Model interface for Fortran models - model world_f.f</td>
</tr>
<tr>
<td>Example 15.2</td>
<td>Semi-automated model interface for Fortran models - model world_f_auto.f</td>
</tr>
<tr>
<td>Example 15.3</td>
<td>Model interface for C models - model world_c.c</td>
</tr>
<tr>
<td>Example 15.4</td>
<td>Model interface for C++ models - model world_cpp.cpp</td>
</tr>
<tr>
<td>Example 15.5</td>
<td>Model interface for Python models - model world_py.py</td>
</tr>
<tr>
<td>Example 15.6</td>
<td>Model interface at shell script level - model shell script world_sh.run</td>
</tr>
<tr>
<td>Example 15.7</td>
<td>Semi-automated model interface at shell script level - model shell script world_sh_auto.run</td>
</tr>
<tr>
<td>Example 15.8</td>
<td>Model interface for GAMS models - model gams_model.gms</td>
</tr>
<tr>
<td>Example 15.9</td>
<td>Experiment post-processor user-defined operator module – operator matmul_f</td>
</tr>
<tr>
<td>Example 15.10</td>
<td>Experiment post-processor user-defined operator module – operator matmul_c</td>
</tr>
<tr>
<td>Example 15.11</td>
<td>ASCII compliant experiment post-processor result import interface</td>
</tr>
</tbody>
</table>
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  
Mc Clelland-Bantam, Toronto, 1995, p. 19
Executive Summary

SimEnv is a multi-run simulation environment that focuses on model evaluation and usage mainly for quality assurance matters and scenario analyses using sampling techniques. Interfacing models to the simulation environment is supported for a number of programming languages by minimal source code modifications and in general at the shell script level. Pre-defined experiment types are the backbone of SimEnv, enabling experimenting with numerical parameter, initial value, or driving forces adjustments of the model. The resulting multi-run experiment can be performed sequentially or in parallel. Interactive experiment post-processing makes use of built-in operator, optionally supplemented by user-defined and composed operators and applies operator chains on model output and reference data. Result output functions generated during post-processing can be evaluated within SimEnv with advanced visualization techniques.

Simulation is one of the cornerstones for research. The aim of the SimEnv project is to develop a toolbox oriented simulation environment that enables 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 coordinated 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 targets $t$ which represent drivers, parameters, boundary and initial values of the model $M$ a sample is generated before simulation and the targets $t$ are re-adjusted numerically before each single simulation run during the experiment. Each experiment results in a sequence of model outputs over the single runs for selected state variables $z$ dependent on the target adjustments of the model $M$. Model outputs can be processed and evaluated across the run ensemble specifically after simulation.

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

- **Behavioural analysis**
  Inspection of the model’s behaviour in a space spanned from targets $t$ with discrete numerical adjustments and a flexible inspection strategy for the whole space.
  For model verification, numerical validation, deterministic error analysis, deterministic control design, scenario analysis and spatial patch model applications.

- **Monte Carlo analysis**
  Perturbations of targets $t$ according to probability density functions. Determination of moments, confidence intervals and heuristic probability density functions for $z$ in the course of 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 targets $t$. Is performed by finite difference derivative approximations from $M$.
  For numerical validation purposes, model analysis, sub-model sensitivity.

- **Optimization**
  Iterative determination of optimal targets $t$ for a cost functions derived from $z$ by a simulated annealing method.
  For model validation (system - model comparison), control design, decision making.

SimEnv makes use of modern IT concepts. Model preparation for interfacing them to SimEnv is based on minimal source code manipulations by implementing interface functions into Fortran-, C/C++-, Python- or GAMS-model source code for target adjustments and model output. Additionally, an interface at shell script level is available.

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

Experiment specific model output post-processing enables navigation in the experiment - model output space and interactive filtering of model output and reference data by application of operator chains. SimEnv supplies built-in operators and enables specification of user-defined and composed operators.
Result evaluation is dominated by application of pre-formed visualization modules. 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.

**SimEnv key features:**

- Available at 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 targets (parameters, initial values, drivers, ...) to experiment with and a strategy how to sample the target space have to be specified.
- Simple model interface to the simulation environment: Model interface functions allow mainly to re-adjust an experiment target 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 and Python: “include per experiment target 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 is a special interface for GAMS models.
- Support of distributed models: Independently on the kind distributed model components are interfaced to SimEnv and among each other the total model can be run within SimEnv.
- Parallelization of the experiment: This is a prerequisite for a lot of simulation tasks.
- Operator-based experiment post-processing: Chains of built-in, user-defined and composed operators enable interactive experiment post-processing based on experiment model output and reference data including general purpose and experiment specific operators. There is a simple interface to write user-defined and to derive composed operators.
- Graphical experiment evaluation: For post-processed model output
- Support of standard data formats: Output from the model as well from the post-processor can be stored in NetCDF or IEEE compliant binary format.

---

**Fig. 0.1** SimEnv system design
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

<table>
<thead>
<tr>
<th>Character / string</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; ... &gt;</td>
<td>angle brackets enclose a placeholder for a string</td>
</tr>
<tr>
<td>{ ... }</td>
<td>braces enclose an optional element</td>
</tr>
<tr>
<td>[ ...</td>
<td>...</td>
</tr>
<tr>
<td>' ... '</td>
<td>single quotation marks enclose a keyword or sub-keyword from user-defined files</td>
</tr>
<tr>
<td>&quot; ... &quot;</td>
<td>double quotation marks enclose the string-value of a sub-keyword from user-defined files</td>
</tr>
<tr>
<td>&lt;nil&gt;</td>
<td>stands for the empty string (nothing)</td>
</tr>
<tr>
<td>monospace</td>
<td>indicates SimEnv example code</td>
</tr>
</tbody>
</table>

Tab. 1.1 Document conventions

Tab. 1.2 summarizes the main placeholders used in this document.

<table>
<thead>
<tr>
<th>Placeholder</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;directory&gt;</td>
<td>path to a file directory</td>
</tr>
<tr>
<td>&lt;file_name&gt;</td>
<td>name of a data file</td>
</tr>
<tr>
<td>&lt;GAMS_model&gt;</td>
<td>name of a GAMS model to start a SimEnv service with</td>
</tr>
<tr>
<td>&lt;int_val&gt;</td>
<td>integer value</td>
</tr>
<tr>
<td>&lt;model&gt;</td>
<td>model name to start a SimEnv service with</td>
</tr>
<tr>
<td>&lt;real_val&gt;</td>
<td>real value in integer, fixed point (e.g., -1.234) or floating point (scientific) notation</td>
</tr>
<tr>
<td>&lt;res&gt;</td>
<td>integer experiment post-processor output file number 1, 2, ..., 99</td>
</tr>
<tr>
<td>&lt;res_char&gt;</td>
<td>character experiment post-processor output file number 01, 02, ..., 99</td>
</tr>
<tr>
<td>&lt;run&gt;</td>
<td>integer single run number 0, 1, ... within an experiment</td>
</tr>
<tr>
<td>&lt;run_char&gt;</td>
<td>character single run number 000000, 000001, ... within an experiment</td>
</tr>
<tr>
<td>&lt;sep&gt;</td>
<td>sequence of white spaces as item separators in user-defined and related files</td>
</tr>
<tr>
<td>&lt;string&gt;</td>
<td>any string</td>
</tr>
<tr>
<td>&lt;target_def_val&gt;</td>
<td>default value of a target as defined in &lt;model&gt;.edf</td>
</tr>
<tr>
<td>&lt;target_name&gt;</td>
<td>name of a target to experiment with as defined in &lt;model&gt;.edf</td>
</tr>
<tr>
<td>&lt;val_list&gt;</td>
<td>list of values in explicit or implicit notation according to Tab. 11.6</td>
</tr>
</tbody>
</table>

For post-processor operator descriptions only

| arg | general numerical argument (operand) |
| int_arg | integer constant argument (operand) ≥ 0 |
| real_arg | real (float) constant argument (operand) |
| char_arg | character argument (operand), enclosed in single quotation marks |

Tab. 1.2 Main placeholders in this document
1.2 Example Layout

All examples in this document but that for the GAMS model (see Section 5.5 on page 30) refer to a hypothetical global simulation model world. It is to describe dynamics of atmosphere and biosphere at the global scale over 200 years. Lateral (latitudinal and longitudinal) model resolution differs for different model implementations (see below), temporal resolution is at decadal time steps. Additionally, atmosphere is structured vertically into levels.

The model world is assumed to map lateral and vertical (level) fluxes and demands that’s why for computing state variables for the whole globe.

The model world is a generic model. Model implementation in several programming languages results in models world_<lng> where <lng> is an identifier for the programming language (and the lateral model resolution).

In the model gridcell_f state variables are calculated for one grid cell (one single latitude-longitude constellation) without consideration of lateral fluxes.

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

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

With this SimEnv release the following model implementations are distributed:

<table>
<thead>
<tr>
<th>Model (“auto” in name = semi-automated model interface)</th>
<th>Model interface example for language &lt;lng&gt;</th>
<th>Resolution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lateral: lat x lon</td>
<td>vertical: number of levels</td>
</tr>
<tr>
<td>world_f</td>
<td>4 x 4</td>
<td>4: 1, 7, 11, 16</td>
</tr>
<tr>
<td>world_c</td>
<td>C</td>
<td>4: 1, 7, 11, 16</td>
</tr>
<tr>
<td>world_cpp</td>
<td>C++</td>
<td>4: 1, 7, 11, 16</td>
</tr>
<tr>
<td>world_py</td>
<td>Python</td>
<td>4: 1, 7, 11, 16</td>
</tr>
<tr>
<td>world_sh</td>
<td>Shell script level</td>
<td>4: 1, 7, 11, 16</td>
</tr>
<tr>
<td>world_f_1x1</td>
<td>Fortran</td>
<td>1 x 1</td>
</tr>
<tr>
<td>world_f_05x05</td>
<td>Fortran</td>
<td>0.5 x 0.5</td>
</tr>
<tr>
<td>world_f_auto</td>
<td>Fortran</td>
<td>4 x 4</td>
</tr>
<tr>
<td>world_sh_auto</td>
<td>Shell script level</td>
<td>4 x 4</td>
</tr>
<tr>
<td>gridcell_f</td>
<td>Fortran</td>
<td>without, implicitly by experiment as 4 x 4</td>
</tr>
</tbody>
</table>

Examples are generally placed in grey-shaded boxes. Examples that are available from the example directory $SE_HOME/../examples 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 (see Tab. 10.3).

Example 1.1 General example layout in the User Guide
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 139.
- Include into the file $HOME/.profile
  
  \[
  \text{export SE\_HOME=/usr/local/simenv/bin} \\
  \text{export PATH=$SE\_HOME:$PATH}
  \]

  to set the SimEnv home directory and execute at the operating system prompt

  \[
  .\ \text{HOME/\_profile}
  \]

- Change to a directory with full access permissions. This is the SimEnv current workspace.
- Start
  
  \[
  \text{simenv.hlp}
  \]

  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 4:
  
  \[
  \langle\text{lng}\rangle = \begin{array}{ll}
  \text{f} & \text{for Fortran} \\
  \text{c} & \text{for C} \\
  \text{cpp} & \text{for C++} \\
  \text{py} & \text{for Python} \\
  \text{sh} & \text{for shell script level}
  \end{array}
  \]

  For a GAMS model example check Section 5.7 on page 33.
- Start
  
  \[
  \text{simenv.cpy world\_<lng>}
  \]

  to copy the model world_<lng> model and experiment related files to the current workspace.
- Copy the file world.edf_c to world_<lng>.edf
- Check
  
  - The SimEnv configuration file world_<lng>.cfg
  - The model output description file world_<lng>.mdf
  - The model world_<lng>.<lng>
  - The model wrap shell script world_<lng>.run
  - The experiment description file world_<lng>.edf
  - The post-processing input file world.post_c

  for general SimEnv configurations available model output variables implementation of the model wrapping the model executable experiment definition post-processor result sequence
- Start
  
  \[
  \text{simenv.cpl world\_<lng> -1 world.post_c}
  \]

  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)
  - Model output post-processing will be started for this experiment
    - With the post-processing input file world_post_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
  
  (*)
- 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  { <run> } }

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

to visualize output from the latest post-processing session world_<lng>.res<res_char>.nc or that
with number <res_char> with the highest two-digit number <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>.lng and/or
  world_<lng>.run
  To experiment with other models replace world_<lng> by <model> as a placeholder for the name of any
other model.

(*)  To get access permission for the visualization server check in Section 10.2 on page 109 the SimEnv
service
  simenv.key  <user_name>
## 3 Version 1.22

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

### 3.1 What is New?

<table>
<thead>
<tr>
<th>Type</th>
<th>Check / see</th>
<th>On page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>update / new</td>
<td>Section 6.1</td>
<td>43</td>
<td>Experiment preparation and performance: Unification and simplification of target adjustments. Target adjustment values are sampled to <code>&lt;model&gt;.edf_adj</code> as specified in <code>&lt;model&gt;.edf</code> and are modified within <code>simenv_get_*</code> by the target default value in a unique manner according to the target type. Special rules for Monte Carlo and local sensitivity analysis are unnecessary furtheron. Introduction of an additional adjustment type = relative: adjusted target value = (1. + <code>&lt;adj_val&gt;</code> * <code>&lt;target_def_value&gt;</code>).</td>
</tr>
<tr>
<td>new</td>
<td>Section 7.3</td>
<td>55</td>
<td>Experiment performance controlled by LoadL: Experiment single runs now can be distributed to all processors of a job class, exploiting in the course of the experiment the free processors of the class in an optimal manner.</td>
</tr>
<tr>
<td>new</td>
<td>Section 8.8</td>
<td>102</td>
<td>Wildcard operands in the experiment post-processor: Use wildcard operands to perform a result expression for all model output variables and/or experiment targets.</td>
</tr>
<tr>
<td>update</td>
<td>Section 7.2</td>
<td>54</td>
<td>SimEnv runs now in the Bourne shell <code>sh</code>: Update the first line in <code>&lt;model&gt;.run</code> and <code>&lt;model&gt;.rst</code> accordingly.</td>
</tr>
<tr>
<td>update</td>
<td>Section 7.3</td>
<td>55</td>
<td>SimEnv runs now in the Bourne shell <code>sh</code>: Update the first line in <code>&lt;model&gt;.run</code> and <code>&lt;model&gt;.rst</code> accordingly.</td>
</tr>
<tr>
<td>update</td>
<td>Section 10.8</td>
<td>118</td>
<td>The whole operating system environment necessary for SimEnv is set within SimEnv. The dot script <code>simenv.env</code> is now dispensable. Exception (include in <code>$HOME/.profile</code>): export <code>SE_HOME=/usr/local/simenv/bin</code> (default) export <code>PATH=$SE_HOME:$PATH</code> (optional)</td>
</tr>
<tr>
<td>update</td>
<td>Tab. 11.3</td>
<td>120</td>
<td>Operating system environment variables now can be used when specifying directories in user-defined files and as arguments in post-processing operators.</td>
</tr>
<tr>
<td>update</td>
<td>Tab. 10.4</td>
<td>111</td>
<td>Files renamed: in <code>$SE_HOME</code>: `simenv_mod_inc.[ f</td>
</tr>
</tbody>
</table>

**Tab. 3.1** SimEnv changes in Version 1.22
Upgrade type | Upgrade action
---|---
mandatory | Update source code of models (Fortran/C/C++) and user-defined operators: rename include files
mandatory | Update <model>.edf for Monte Carlo and local sensitivity analyses
mandatory | Rename optional job control files and optimization options file
mandatory | Update <model>.run and <model>.rst: change shell
mandatory | Update $HOME/.profile
mandatory | Re-link models interfaced to SimEnv and user-defined operators

**Table 3.2** User actions to upgrade to Version 1.22

### 3.2 Limitations / Problems and Their Workarounds

<table>
<thead>
<tr>
<th>Where Limitation / Problem Workaround</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where</td>
<td>Overall</td>
</tr>
<tr>
<td>Limitation</td>
<td>Current SimEnv technical limitations as specified in Tab. 15.3 on page 140</td>
</tr>
<tr>
<td>Workaround</td>
<td>None</td>
</tr>
<tr>
<td>Where</td>
<td>Overall but visual result evaluation</td>
</tr>
<tr>
<td>Limitation</td>
<td>Without graphical user interface</td>
</tr>
<tr>
<td>Workaround</td>
<td>None</td>
</tr>
<tr>
<td>Where</td>
<td>Experiment performance: Experiment type optimization</td>
</tr>
<tr>
<td>Limitation</td>
<td>Can not be performed in parallel mode</td>
</tr>
<tr>
<td>Workaround</td>
<td>Perform optimization experiment in sequential mode</td>
</tr>
<tr>
<td>Where</td>
<td>Experiment performance: Experiment type optimization</td>
</tr>
<tr>
<td>Limitation</td>
<td>The initial seed for the optimization technique is fixed. That’s why the algorithm results for the same optimization problem always in the same sampled sequence in the target space</td>
</tr>
<tr>
<td>Workaround</td>
<td>None</td>
</tr>
<tr>
<td>Where</td>
<td>Experiment performance: Model output to NetCDF</td>
</tr>
<tr>
<td>Problem</td>
<td>Check on undefined model output results in noticably additional CPU-time consumption. Example: to check 8 Mill of real*8 values takes per single run additionally 80 sec for single nc-file model output and 200 sec for common nc-file output.</td>
</tr>
<tr>
<td>Workaround</td>
<td>Specify in &lt;model&gt;.cfg for sub-keyword ‘message_level’ value = “error”</td>
</tr>
<tr>
<td>Where</td>
<td>Experiment post-processing: Optional specification / automated identification of result description and result unit</td>
</tr>
<tr>
<td>Limitation</td>
<td>Not stored to NetCDF result output</td>
</tr>
<tr>
<td>Workaround</td>
<td>Specify IEEE or ASCII result output instead</td>
</tr>
</tbody>
</table>

**Table 3.3** Limitations / problems and their workarounds
### 3.3 Known Bugs and Their Workarounds

<table>
<thead>
<tr>
<th>Where Bug Workaround</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where Bug Workaround</td>
<td>Description</td>
</tr>
<tr>
<td>Where Experiment restart</td>
<td>Model output to a common NetCDF file for the whole experiment</td>
</tr>
<tr>
<td>Bug Model output is not stored</td>
<td></td>
</tr>
<tr>
<td>Workaround Specify IEEE model output or single NetCDF file output in &lt;model&gt;.cfg</td>
<td></td>
</tr>
<tr>
<td>Where Experiment performance:</td>
<td>Model output to NetCDF of distributed models (structure = distributed in &lt;model&gt;.cfg)</td>
</tr>
<tr>
<td>Bug May not store all model output</td>
<td></td>
</tr>
<tr>
<td>Workaround Specify IEEE model output in &lt;model&gt;.cfg</td>
<td></td>
</tr>
<tr>
<td>Where Experiment post-processing:</td>
<td>Result output to NetCDF</td>
</tr>
<tr>
<td>Bug simenv.res &lt;model&gt; append does not work</td>
<td></td>
</tr>
<tr>
<td>Workaround Specify result output to IEEE or ASCII instead</td>
<td></td>
</tr>
<tr>
<td>Where Experiment post-processing:</td>
<td>Behavioural analysis / result output to NetCDF</td>
</tr>
<tr>
<td>Bug When applying the operator behav non-monotonic and monotonously decreasing target adjustments are transferred to the NetCDF output file in a wrong manner.</td>
<td></td>
</tr>
<tr>
<td>Workaround Specify only monotonously increasing target adjustments in &lt;model&gt;.edf or specify IEEE and/or ASCII post-processor output in &lt;model&gt;.cfg</td>
<td></td>
</tr>
</tbody>
</table>

*Tab. 3.4* Known bugs and their workarounds
4 Experiment Types

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

4.1 General Approach

SimEnv supplies a set of pre-defined multi-run experiment types, where each type addresses a special multi-run experiment class for performing a simulation model or any algorithm with an input - output transition behaviour.

In the following, the general SimEnv approach will be described for time dynamic simulation models, because this class forms the majority of SimEnv applications. All information can be transformed easily to any other algorithm.

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

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

with

- $ST$: state transition description
- $Z$: state variables’ vector
- $P$: parameter vector
- $X$: input (driving forces) vector
- $Z_0$: initial value vector
- $B$: boundary value vector
- $t$: time
- $\Delta t$: time increment
- $k$: time delay

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

$$Y(t) = OU ( Z(t) , P , X(t) , Z_0 )$$

Model behaviour $Z$ is determined for fixed $k$ and $\Delta t$ by state transition description $ST$, parameters $P$, driving forces $X$, initial values $Z_0$, and boundary values $B$. Manipulating and exploring model behaviour in any sense means changing these four model components. While state transition description $ST$ reflects mainly model structure and is quite complex to change, each component of the driving forces vector $X$ 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 applying for example alternative model structures, sub-structures, processes formulations, resolutions, which are triggered by these values.

Additionally, each component of the driving forces vector $X$ 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 adjustment)
- By parametrizing the shape of a driving force

When this has been done, the model behaviour finally depends only on the parameters $P$, the initial values $Z_0$, and the boundary values $B$. From the methodical point of view there is no difference between parameters, initial values and boundary values, because all are considered as constant during one model run. That
is why in SimEnv all the four model components parameters, drivers, initial values and boundary values are lumped together and the term target stands as a placeholder for them. All targets form the target set T:

\[ T = \{ P, X, Z_0, B \} \]

and

\[ Z = ST(T). \]

In the following,

\[ T_k = (t_1, \ldots, t_k) \quad k > 0 \]

stands for a subset of the target set T that spans up an k-dimensional sub-space of T by selected model targets \((t_1, \ldots, t_k)\) from T and

\[
T_{k,n} = \begin{pmatrix}
  t_{11} & \cdots & t_{1n} \\
  \vdots & \ddots & \vdots \\
  t_{k1} & \cdots & t_{kn}
\end{pmatrix} = (\hat{T}_1, \ldots, \hat{T}_n) \quad k > 0, \ n > 0
\]

stands for a numerical sample for \(T_k\) of size \(n\) and finally for \(k*n\) values representing in any sense the sample space \(T_k\).

In the set of all samples \(T_{k,1}\), \(T_{k,1}\) is the nominal (default) numerical target constellation for the model \(M\).

If \(\{\_\}_n\) denotes the dynamics of the model \(M\) over a sample of size \(n\) then it holds:

\[ \{ Z \}_n = \{ ST(\hat{T}_1), \ldots, ST(\hat{T}_n) \} \]

SimEnv supports different sampling strategies and performance of multi-run experiments where \(m\) targets are readjusted numerically for each of \(n\) single simulation runs. Central goal is to study dependency of the model dynamics on target adjustments. For simulation purposes in SimEnv experimentation with the model \(M\) over \(T_{k,n}\) 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 \(T_{k,n}\).

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

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

The experiment types are described in detail in the following.

The term target was selected as an analogue to experimentation with real systems: Often a target is under investigation to study the change in the real system when the state of the target is modified by the experimenter. Often used synonyms for “target” are “input” and “factor.”

Fig. 4.1  Target space
4.2 Behavioural Analysis

Behavioural analysis uses a deterministic strategy to sample \( T_k \). It is the inspection of the model in the target space \( T_k \) where inspection points are set in a regular and well structured manner.

Behavioural analysis can be interpreted and used in different ways:
- For scenario analysis:
  to show how model behaviour changes with changes of target values
- For numerical validation purposes:
  to determine target 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 target errors
- For a simulation-based control design:
  to determine target values in such a way that a goal function becomes an extreme

\[
\{x\} = T_{2,12}
\]

\( x \) is a sample of size 12 in the 2-dimensional target space
\( T_2 = (t_1, t_2) \)

\( o = T_{2,1} \)

nominal (default) numerical target constellation of model M

**Fig. 4.2** Sample for a behavioural analysis

SimEnv behavioural analysis sampling strategy is a generalization of the one-dimensional case for \( T_1 \), where the model behaviour is scanned in dependence on deterministic adjustments of one target \( t_1 \). The general case for \( T_k \) 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 target 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 target samples per dimension of the scanned target space and from the selected scanning method. An experiment is described by the names of the involved targets, their numerical adjustments and their combination (scanning method). Experiment post-processing can resolve the scanning method again and output results as projections on multi-dimensional target subspaces.

**Fig. 4.3** describes the regular scanning technique by an example. In the left scheme (a) the two-dimensional target space \( T_2 = (p_1, p_2) \) is scanned combinatorially, resulting in \( 4^2 = 16 \) model runs, while the middle scheme (b) represents a parallel scanning of these two targets at the diagonal by \( 1+1+1+1 = 4 \) model runs. The scheme (c) at the right side shows a complex scanning strategy of the 3-dimensional target space \( T_3 = (p_1, p_2, p_3) \) with \( (1+1+1+1)^3 = 12 \) model runs. Each filled dot \( ● \) in Fig. 4.3 correspond to an cross \( x \) in Fig. 4.2 and represents a sample point in the target space and finally a single model run of the experiment.
4.3 Monte Carlo Analysis

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

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

$$M^{(m)}(z) = \int \cdots \int z(T_k)^m \cdot pdf(T_k) \ dt_k$$

with

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

By interpreting the probability density function pdf($T_k$) as the error distribution in the target space $T_k$, 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($T_k$) can be decomposed into independent probability density functions pdf, for all targets $t_i$ of $T_k$.
and the k-dimensional integral is approximated by a sequence of n single simulation runs of the model where
the numerical target values $t_i$ of $t_i, (1 \leq i \leq k, 1 \leq j \leq n)$ are sampled according to the probability density function $pdf$, 

On the basis of these assumptions, the statistical measures in Tab. 4.1 can be computed during performance of an experiment post-processing session from a Monte Carlo analysis with n simulation runs resulting in n realizations $z_1, ..., z_n$ of the model’s state variables $z, z1$ and $z2$:

<table>
<thead>
<tr>
<th>Statistical measure</th>
<th>Definition (*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>minimum</td>
<td>$\text{min}(z) = \min(z_i)$</td>
</tr>
<tr>
<td>maximum</td>
<td>$\text{max}(z) = \max(z_i)$</td>
</tr>
<tr>
<td>sum</td>
<td>$\sum z_i$</td>
</tr>
<tr>
<td>arithmetic mean</td>
<td>$\text{avg}(z) = \frac{\sum z_i}{n}$</td>
</tr>
<tr>
<td>variance</td>
<td>$\text{var}(z) = \frac{\sum (z_i - \text{avg}(z))^2}{n - 1}$</td>
</tr>
<tr>
<td>skewness</td>
<td>$\text{skw}(z) = \frac{\sum (z_i - \text{avg}(z))^3}{n \cdot \left( \frac{\sum (z_i - \text{avg}(z))^2}{n - 1} \right)^{3/2}}$</td>
</tr>
<tr>
<td>kurtosis</td>
<td>$\text{krt}(z) = \frac{\sum (z_i - \text{avg}(z))^4}{n \cdot \left( \frac{\sum (z_i - \text{avg}(z))^2}{n - 1} \right)^2} - 3$</td>
</tr>
<tr>
<td>range</td>
<td>$\text{rng}(z) = \text{max}(z) - \text{min}(z)$</td>
</tr>
<tr>
<td>geometric mean</td>
<td>$\text{avgg}(z) = \left( \prod z_i \right)^{1/n}$</td>
</tr>
<tr>
<td>harmonic mean</td>
<td>$\text{agvh}(z) = \frac{n}{\sum (1/z_i)}$</td>
</tr>
<tr>
<td>weighted mean</td>
<td>$\text{avgw}(z) = \frac{\sum z_i \cdot w_i}{\sum w_i}$ $w : \text{weight}$</td>
</tr>
<tr>
<td>correlation</td>
<td>$\text{cor}(z_1, z_2) = \frac{\sum (z_{1i} - \text{avg}(z_1)) \cdot (z_{2i} - \text{avg}(z_2))}{\sqrt{\sum (z_{1i} - \text{avg}(z_1))^2 \cdot \sum (z_{2i} - \text{avg}(z_2))^2}}$</td>
</tr>
<tr>
<td>covariance</td>
<td>$\text{cov}(z_1, z_2) = \frac{\sum (z_{1i} - \text{avg}(z_1)) \cdot (z_{2i} - \text{avg}(z_2))}{n - 1}$</td>
</tr>
<tr>
<td>linear regression coefficient</td>
<td>$\text{reg}(z_1, z_2) = \frac{\sum (z_{1i} - \text{avg}(z_1)) \cdot (z_{2i} - \text{avg}(z_2))}{\sum (z_{1i} - \text{avg}(z_1))^2}$</td>
</tr>
<tr>
<td>median</td>
<td>$\text{med}(z) = \text{middle value from increasingly ordered } { z_i }$ $\text{mean of the two middle values from } { z_i }$ $(n = \text{odd})$ $(n = \text{even})$</td>
</tr>
<tr>
<td>quantile</td>
<td>$\text{qnt}^{(p)}(z) = \text{that value from increasingly ordered } { z_i }$ which corresponds to a cumulative frequency of $n \cdot p$ $\text{qnt}^{(0.5)}(z) = \text{med}(z)$</td>
</tr>
<tr>
<td>confidence interval bounds</td>
<td>$\text{cnf}^{(\alpha)}(z) = \text{avg}(z) \pm \frac{t_{\alpha,n-1}}{\sqrt{\text{var}(z)/n}}$ $\alpha : \text{level of error}$ $t_{\alpha,n} : \text{significance boundaries of Student distribution}$</td>
</tr>
<tr>
<td>heuristic probability density function</td>
<td>$hgr^{(class)}(z) = \text{number of } z_i \text{ with } \text{class}<em>{\text{min}} \leq z_i &lt; \text{class}</em>{\text{max}}$ class_{\text{min}}, class_{\text{max}}: boundaries of equidistant classes</td>
</tr>
</tbody>
</table>

Tab. 4.1: Statistical measures

(*) indices for sums $\sum$, products $\Pi$ and extremes run from 1 to n: $\sum_{i=1}^{n}, \Pi_{i=1}^{n}, \min_{i=1}^{n}, \max_{i=1}^{n}$
Tab. 4.2 summarizes these probability density functions (Bohr, 1998) that are pre-defined in SimEnv for targets to be perturbed. Additionally, SimEnv offers to import random number samples in the course of experiment preparation.

<table>
<thead>
<tr>
<th>Distribution</th>
<th>Short-cut</th>
<th>Probability density function pdf</th>
<th>Distribution parameters</th>
</tr>
</thead>
</table>
| uniform      | U(a,b)    | \( \begin{align*}
    pdf(x) &= \frac{1}{b-a} & \text{if } x \in [a,b] \\
    pdf(x) &= 0 & \text{otherwise}
  \end{align*} \) | a: lower boundary \\
b: upper boundary > a \\
it is: mean = (a+b) / 2 \\
standard deviation = \( \sqrt{(b-a)^2 / 12} \) |
| normal       | N(\(\mu\),\(\sigma^2\)) | \( \begin{align*}
    pdf(x) &= \frac{1}{\sqrt{2\pi} \sigma} \exp\left( -\frac{(x-\mu)^2}{2\sigma^2} \right) & \text{if } x > 0 \\
    pdf(x) &= 0 & \text{otherwise}
  \end{align*} \) | \(\mu\): mean \\
\(\sigma\): standard deviation > 0 |
| lognormal    | L(\(\mu\),\(\sigma^2\)) | \( \begin{align*}
    pdf(x) &= \frac{1}{x\sigma\sqrt{2\pi}} \exp\left( -\frac{\ln(x-\mu)^2}{2\sigma^2} \right) & \text{if } x > 0 \\
    pdf(x) &= 0 & \text{otherwise}
  \end{align*} \) | \(\mu\): mean > 0 |
| exponential  | E(\(\mu\)) | \( \begin{align*}
    pdf(x) &= \frac{1}{\mu} \exp\left( -\frac{x}{\mu} \right) & \text{if } x > 0 \\
    pdf(x) &= 0 & \text{otherwise}
  \end{align*} \) | \(\mu\): mean > 0 |

**Tab. 4.2** Probability density functions

The number of runs to be performed during a Monte Carlo analysis has to be specified. An experiment is described by the targets involved in the analysis, their distribution and the appropriate distribution parameters.

### 4.4 Local Sensitivity Analysis

Local sensitivity analysis uses a deterministic sampling strategy in \(\epsilon\)-neighbourhoods of the numerical default constellation \(T_{k,1}\) of the model \(M\). For each target \(t\) from the nominal target constellation \(T_{k,1}\) and each \(\epsilon_j\) from the \(\epsilon\)-neighbourhoods (\(\epsilon_1\), ..., \(\epsilon_m\)) two members (\(t_1\), ..., \(t_i\), \(t\pm\epsilon_j\), \(t_{i+1}\), ..., \(t_k\)) of the resulting sample are generated. The sample size \(n\) is given by \(2^m n^k\). 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 target \(t\) is the partial derivative of \(z\) after \(t \frac{\delta z}{\delta t}\). 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 around the nominal target constellation.

Local sensitivity measures as well as measures which reflect model output linearity and/or symmetry nearby \(T_{k,1}\) can be used for localizing modification-relevant model parts as well as control-sensitive targets 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. 4.3.
Accordingly, local measures of the model with respect to a target are always expressed as a measure of a model’s state variable \( z \), usually at a selected time step within a surrounding neighborhood \( \varepsilon \) of a target value \( t \). That is why the conclusions drawn from a local sensitivity analysis are only valid locally at \( T_{k,1} \) with respect to the whole target space \( T_k \). Additionally, local measures only describe the influence of one target \( t_i \) from the whole vector \( T_k \) on the model’s dynamics.

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

The absolute measures are best suited to compare the influence of different targets \( \{t\} \) on the same state variable \( z \) while due to their normalization factor the relative measures enable comparison of the influence of one target \( t \) on different state variables \( \{z\} \).

From the local measures of table Tab. 4.3 additional measures can be derived on demand, e.g., \( \text{abs}(\text{sym}_\text{abs}(z, \varepsilon)) \).

A local sensitivity experiment is described by the names of the targets \( t \) to be involved and the increments \( \varepsilon \). The number of runs for the experiment results from the number of targets and increments: two runs per tar-
get for each increment plus one run with the default values of the targets. Local sensitivity functions are calculated during experiment post-processing.

4.5 Optimization

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

$$F(Z) = F(ST(T_k))$$

that depends on model’s state variables $Z$ and consequently on the experiment targets $T_k = (t_1, ..., t_k)$:

$$\text{minimize} \quad F(t_1, ..., t_k)$$

$$\text{subject to} \quad t_{i_{\min}} \leq t_i \leq t_{i_{\max}} \quad \text{for} \quad i = 1, ..., k$$

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 targets 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 (check Section 6.5 and Chapter 8). The value of the cost function is calculated directly after the current single run has been performed.

SimEnv uses a gradient free optimization approach that is called “Simulated Annealing” and is a generalization of a Monte Carlo method for examining the state equations of n-body systems. The concept is based on the manner in which metals recrystallize 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

$$p = \exp(-dE/(k_B \times \text{Temp}))$$

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

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

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

The major difficulty (art) in implementation of a simulated annealing algorithm is that there is no obvious analogy for the temperature Temp with respect to a free parameter in the optimization problem. Furthermore, avoidance of entrainment in local minima (quenching) is dependent on the "annealing schedule", that is, the choice of initial temperature, how many iterations are performed at each temperature, and how much the temperature is decremented at each step as cooling proceeds (after Gray et al., 1997). Ideally, when local optimization methods are trapped in a poor local minimum, simulated annealing can 'climb' out.
The algorithm applied in SimEnv is a very fast simulated re-annealing method, named Adaptive Simulated Annealing ASA (Ingber 2004, Ingber 1989 and Ingber 1996). For the above stated probability $p$ the term $k_B \cdot \text{Temp}$ is chosen as

$$k_B \cdot \text{Temp} = \text{Temp}_0 \cdot \exp(-c^*k^{1/m})$$

where $k$ is the annealing time.

The ASA schedule is much faster than Boltzmann annealing, where $k_B \cdot \text{Temp} = \text{Temp}_0 / \ln(k)$ and faster than fast Cauchy annealing, where $k_B \cdot \text{Temp} = \text{Temp}_0 / k$. With the ASA method the global minimum of a nonlinear non-convex cost function $F$ over an $m$-dimensional bounded target space $T_k$ is determined.

Fig. 4.6 Part of a sample for an optimization experiment, generated during the experiment
5 Model Interface

To use any model within SimEnv it has to be interfaced to the simulation environment. SimEnv offers easy
coupling techniques at programming language and shell script level. While at language level SimEnv func-
tion calls have to be implemented into model source code to adjust experiment targets, i.e. model param-
ters, initial values or boundary values of the current single run out of the run ensemble numerically 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 simula-
tion environment the variables of the model to be output during experiment performance and to be potentially
processed during experiment post-processing have to be declared in the model output description file
<model>.mdf. Additionally, the model itself has to be wrapped into a shell script <model>.run.
Model interfacing is related to transferring adjusted numerical values of model targets under investigation
from the simulation environment to the model and to transferring model output variables under investigation
from the model to the simulation environment for later experiment post-processing. Interfacing is supported
at the programming language level for C/C++, Fortran, Python, and GAMS programming languages, the
model is implemented in and at shell script level.

5.1 General Approach

SimEnv model interface has to supply a link between the simulation environment and the model and has to
address two aspects:
For each single run from the run ensemble
• All numerical adjustments of experiment targets as defined in the experiment description file
  <model>.edf (check Section 6.1) have to be associated to the corresponding model entities (parame-
ters, initial or boundary values, drivers) and these entities have to be modified numerically in the model
  according to the specified adjustments.
• All model output variables as defined in the model output description file <model>.mdf (check 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.

Realisation 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. Gener-
ally speaking,
• Every experiment target and
• Every model output variable
demand one additional SimEnv function call in the model source code. According to Tab. 5.1 model inter-
face functions are generic.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>simenv Ini &lt;lng&gt;</td>
<td>open model coupling interface</td>
</tr>
<tr>
<td>simenv get_&lt;lng&gt;</td>
<td>associate a model source code entity with an experiment target (parameter / initial value / boundary value) from &lt;model&gt;.edf and get the target adjustment</td>
</tr>
<tr>
<td>simenv get run_&lt;lng&gt;</td>
<td>get the current single run number of the run ensemble</td>
</tr>
<tr>
<td>simenv put_&lt;lng&gt;</td>
<td>associate a model source code entity with a model output variable from &lt;model&gt;.mdf and output it to SimEnv data structures</td>
</tr>
<tr>
<td>simenv slice_&lt;lng&gt;</td>
<td>enable slicing, i.e., a repetitively partial output of model output variables.</td>
</tr>
<tr>
<td>simenv end_&lt;lng&gt;</td>
<td>close model coupling interface</td>
</tr>
</tbody>
</table>

Tab. 5.1 Generic SimEnv interface functions
(for <lng> check Tab. 5.2)
The function simenv_slice_<lng> announces output of a slice of the data of a defined model output variable. This is good for models with multi-dimensional variables where at least one dimension is omitted in the state variable declaration in the model the source code because the dynamics for this dimension is calculated in place (e.g., time). The assigned variable then has a lower dimensionality than the corresponding variable in the model output description file. Nevertheless, the simenv_slice_<lng>-function ensures that model output over the omitted dimension can be handled in experiment post-processing in common.

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

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

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

A regular matching between the model output description file and the used SimEnv interface functions in the model source code as well as the above exceptions are reported to the interface log-file <model>.mlog (check Tab. 10.7).

Native model output does not influence performance of the model in SimEnv and there is no necessity to disable this output for SimEnv. The user only has to ensure that for a experiment control by the load leveler LoadL the outputs of different single runs do not conflict with each other. Normally, this can be ensured by performing each single run in a special run-related sub-directory (check Example 15.6). Native user model output to the terminal is redirected during the experiment to the log-file <model>.nlog.

For running an interfaced model outside SimEnv there are dummy SimEnv libraries to link / run the model with. They ensure the same model dynamics as before interfacing the model to SimEnv (check Section 5.10).

Currently, there are SimEnv interfaces for Fortran, C/C++, Python and GAMS models. Additionally, there is an interface implementation at shell script level. Mixed language models as well as distributed models (check Section 5.9) can be run with SimEnv.

<table>
<thead>
<tr>
<th>&lt;lng&gt;</th>
<th>for model source code</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>C/C++</td>
</tr>
<tr>
<td>f</td>
<td>Fortran</td>
</tr>
<tr>
<td>py</td>
<td>Python</td>
</tr>
<tr>
<td>sh</td>
<td>Shell script level</td>
</tr>
</tbody>
</table>

Tab. 5.2 Language suffices for SimEnv interface functions (for the GAMS interface check Section 5.7)
Fig. 5.1  Conceptual scheme of the model interface for C/C++/Fortran/Python
5.2 Grid and Coordinate Assignments to Variables

To each variable

- Dimensionality $\dim(variable)$
- Extents $\text{ext}(variable,i)$ with $i=1,\ldots,\dim(variable)$
- Coordinates $\text{coord}(variable,i)$ with $i=1,\ldots,\dim(variable)$

are assigned. 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. Coordinates have a name and from all coordinate values the coordinate is defined for a subset is assigned to the extent of the dimension of the variable. Variables of dimensionality 0 do not have a coordinate assignment.

A variable of dimensionality $n$ corresponds with an $n$-dimensional array, a variable of dimensionality 0 is a scalar.

![Grid types](image)

**Fig. 5.2** Grid types

Additionally, coordinate axes are defined. Each coordinate axis a strictly monotonic sequence of coordinate values, a description and a unit is assigned to. For reasons of simplification in experiment post-processing coordinate axes are assumed as curvilinear.

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

Since coordinate axes can be assigned to model output variable dimensions in a flexible manner, model output variables can exist on the same coordinate system or completely or partially disjoint coordinate systems.
5.3 Model Output Description File <model>.mdf

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

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>&lt;nil&gt;</td>
<td>descr</td>
<td>o</td>
<td>any</td>
<td>&lt;string&gt;</td>
<td>model output description</td>
</tr>
<tr>
<td>coordinate</td>
<td>coordinate name</td>
<td>descr</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>coordinate axis description</td>
</tr>
<tr>
<td></td>
<td>(&lt;co_name&gt;)</td>
<td>unit</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>coordinate axis unit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values</td>
<td>m</td>
<td>1</td>
<td>&lt;val_list&gt;</td>
<td>strictly monotonic sequence of coordinate values &lt;co_vals&gt; (for syntax see Tab. 11.6)</td>
</tr>
<tr>
<td>variable</td>
<td>&lt;variable_name&gt;</td>
<td>descr</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>variable description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unit</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>variable unit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
<td>m</td>
<td>1</td>
<td>see Tab. 5.4</td>
<td>variable type in the simulation model</td>
</tr>
<tr>
<td></td>
<td></td>
<td>coords</td>
<td>c1</td>
<td>1</td>
<td>&lt;co_name1&gt;, &lt;co_name2&gt;</td>
<td>assigns a coordinate axis by its name to each dimension of the variable. Determines in this way implicitly the dimensionality n of the variable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>coord_extents</td>
<td>c2</td>
<td>1</td>
<td>&lt;co_val1&gt;, &lt;co_val2&gt;</td>
<td>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.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>index_extents</td>
<td>c1</td>
<td>1</td>
<td>&lt;in_val1&gt;, &lt;in_val2&gt;</td>
<td>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.</td>
</tr>
</tbody>
</table>

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

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

<model>.mdf is an ASCII file that holds this information. It follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

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

- For the description of line type check Tab. 11.4 on page 121.
- Coordinate and variable names must differ from target names in experiment description (see Section 6.1) and from built-in and user-defined operator names for experiment post-processing (see Section 8.5.4).
- Assignment of coordinate axes to variable dimensions and consequently of a grid to a variable is only valid for experiment post-processing. Normally, the simulation model itself will also exploit the same grid.
structure. Nevertheless, the grid structures of the model are defined autonomously in the model in an explicit or implicit manner and do only correspond with the grid structure in the model output description file symbolically.

- Model output variables with dimensionality 0 are not assigned to a coordinate axis.
- The values of a coordinate have to be ordered in a strictly monotonic sequence. They may be non-equidistant and may be ordered in a decreasing sequence.
- With the sub-keyword ‘coord_extents’ only a portion of coordinate values of a coordinate axis can be assigned to a dimension of a variable. This portion is addressed by its begin and end value <co_val_i1> and/or <co_val_i2>. The number of coordinates values of the portion has to be greater than 1.
  - <co_val_i1> < co_val_i2> for strictly increasing values of coordinates
  - <co_val_i1> > co_val_i2> for strictly decreasing values of coordinates
- With the sub-keyword ‘index_extents’ portions of variables are made addressable during SimEnv experiment post-processing. In the same way multi-dimensional variables are equipped with indices in the simulation model they also have an index description in the model output description file for purposes of experiment post-processing. It is advisable, that these two descriptions coincide. The index range is described by a start and an end integer value index <in_val_i1> and/or <in_val_ext_i2>.
  - The index set is a strictly increasing, equidistant set of integer values with an index increment of 1, <in_val_i1> ≤ <in_val_i2>
  - <in_val_i1> ≤ 0 is possible.
- Coordinate values <co_val> and index values <in_val> are assigned in a one-to-one manner.
- For multi-dimensional variables that do not exist on an assigned grid completely or partially, simply assign formal coordinate axes to.
- Specify at least one model output variable in <model>.mdf

<table>
<thead>
<tr>
<th>SimEnv data type (synonyms)</th>
<th>Description</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>int*1</td>
<td>1 byte integer</td>
</tr>
<tr>
<td>short</td>
<td>int*2</td>
<td>2 bytes integer</td>
</tr>
<tr>
<td>int</td>
<td>int*4</td>
<td>4 bytes integer</td>
</tr>
<tr>
<td>float</td>
<td>real*4</td>
<td>4 bytes real</td>
</tr>
<tr>
<td>double</td>
<td>real*8</td>
<td>8 bytes real</td>
</tr>
</tbody>
</table>

Tab. 5.4 SimEnv data types

For the following example of a model output description file and the assigned grids for model output variables check Example 1.1 on page 4:

<table>
<thead>
<tr>
<th>general</th>
<th>descr</th>
<th>World with a resolution of</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>descr</td>
<td>4° lat x 4° lon x</td>
</tr>
<tr>
<td>general</td>
<td>descr</td>
<td>4 levels x 20 time steps</td>
</tr>
<tr>
<td>general</td>
<td>descr</td>
<td>Data centred per lat-lon cell</td>
</tr>
<tr>
<td>general</td>
<td>descr</td>
<td>This file is valid for all models</td>
</tr>
<tr>
<td>general</td>
<td>descr</td>
<td>world_[ f</td>
</tr>
<tr>
<td>coordinate lat</td>
<td>descr</td>
<td>geographic latitude</td>
</tr>
<tr>
<td>coordinate lat</td>
<td>unit</td>
<td>deg</td>
</tr>
<tr>
<td>coordinate lat</td>
<td>values</td>
<td>equidist_end 88(-4)-88</td>
</tr>
<tr>
<td>coordinate lon</td>
<td>descr</td>
<td>geographic longitude</td>
</tr>
<tr>
<td>coordinate lon</td>
<td>unit</td>
<td>deg</td>
</tr>
<tr>
<td>coordinate lon</td>
<td>values</td>
<td>equidist_end -178(4)178</td>
</tr>
<tr>
<td>coordinate level</td>
<td>descr</td>
<td>atmospheric vertical level</td>
</tr>
<tr>
<td>coordinate level</td>
<td>unit</td>
<td>level no</td>
</tr>
<tr>
<td>coordinate level</td>
<td>values</td>
<td>list 1,7,11,16</td>
</tr>
</tbody>
</table>
coordinate time descr time in decades
countinate time unit 10 years
countinate time values equidist_nmb 1(1)20

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

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

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

variable bios_g type int

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

Example 5.1  Model output description file <model>.mdf

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 adjust experiment targets 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.
<table>
<thead>
<tr>
<th>Function name</th>
<th>Function description</th>
<th>Arguments / function value</th>
<th>Argument / function value</th>
</tr>
</thead>
<tbody>
<tr>
<td>simenv_ini_ [f</td>
<td>c] ( )</td>
<td>initialize model coupling interface</td>
<td>integer*4 simenv_ini_ [f</td>
</tr>
<tr>
<td></td>
<td>Perform always as the first SimEnv function in the model. Alternatively include &lt;model&gt;_ [f</td>
<td>c].inc for semi-automated model interface</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 2 I/O error for model output file</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 3 error memory allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 4 I/O error for &lt;model&gt;.edf_bin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 5 I/O error for &lt;model&gt;.mdf_bin</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 6 I/O error for &lt;model&gt;.edf_adj</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 7 wrong single run number</td>
</tr>
<tr>
<td>simenv_get_ [f</td>
<td>c] (target_name, target_def_val, target_adj_val)</td>
<td>get the numerical adjustment in the current single run for the target to be experimented with</td>
<td>character*() target_name (input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real*4 target_def_val (input)</td>
<td>nominal / default (non-adjusted) target value. If target_name is not defined in &lt;model&gt;.edf then target_adj_val is set to target_def_val</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real*4 target_adj_val (output)</td>
<td>adjusted target value</td>
</tr>
<tr>
<td>simenv_get_run_ [f</td>
<td>c] (run_int, run_char)</td>
<td>get run number of the current run as an integer value and a character string</td>
<td>character*6 run_char (output)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 run_int (output)</td>
<td>current run number</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 0 ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 1 target_name undefined: target_adj_val := target_def_val</td>
</tr>
<tr>
<td>simenv_put_ [f</td>
<td>c] (var_name, field)</td>
<td>output model results to native SimEnv output file(s)</td>
<td>character*() var_name (input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>dimension field(...), type according to &lt;model&gt;.mdf (input)</td>
<td>data of variable var_name to be stored as simulation results</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 simenv_put_ [f</td>
<td>c] (function value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 0 ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 1 var_name undefined</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 2 I/O error for model output file</td>
</tr>
</tbody>
</table>
Tab. 5.5  
Model interface functions for Fortran and C/C++ models

- Make sure consistency of type and dimension declarations between the model output variables in model source code and the corresponding variable declarations in the model output description file <model>.mdf.
- Model output variables that are not output completely or partially within the user model are handled in experiment post-processing as their corresponding nodata-values (see Tab. 10.12).
- Application of simenv_slice_* 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 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++. Additionally, these include file declare for the semi-automated model interface (see Section 5.8) auxiliary variables. For the contents of the include files check Tab. 10.5.
- Apply the shell script
  simenv_mod_[f | c | cpp].lnk <model_name>
From the SimEnv home directory to compile and link an interfaced model.
- User models implemented in C/C++ or Fortran have to be linked with the following libraries to interface them to the simulation environment
  - $SE_HOME/libsimenv.a
  - /usr/local/lib/libnetcdf.a
- Tab. 15.12 lists the additionally used symbols when interfacing a Fortran or C/C++ model to SimEnv.
- In
  - Example 15.1 on page 144 the model world_f.f
  - Example 15.3 on page 147 the model world_c.c
  - Example 15.4 on page 149 the model world_cpp.cpp
are explained.
## 5.5 Model Interface for Python Models

<table>
<thead>
<tr>
<th>Function name</th>
<th>Function description</th>
<th>Arguments / function value</th>
<th>Argument / function value description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>simenv_init_py()</code></td>
<td>initialize model coupling interface</td>
<td>string <code>ini_py</code> (function value)</td>
<td>return code of the spawn function for a SimEnv executable</td>
</tr>
<tr>
<td><code>simenv_get_py(target_name, target_def_val)</code></td>
<td>get the numerical adjustment in the current single run for the target to be experimented with</td>
<td>string <code>target_name</code> (input) float <code>target_def_val</code> (input) float <code>get_py</code> (function value)</td>
<td>name of the target in <code>&lt;model&gt;.edf</code> nominal / default (non-adjusted) target value. If <code>target_name</code> is not defined in <code>&lt;model&gt;.edf</code> then <code>target_adj_val</code> is set to <code>target_def_val</code> adjusted target value <code>target_adj_val</code></td>
</tr>
<tr>
<td><code>simenv_get_run_py()</code></td>
<td>get the run number of the current run as a character string</td>
<td>string <code>get_run_py</code> (function value)</td>
<td>current run number as string of the length 6 with leading zeros. If an error occurred then <code>run_char</code> = '------'</td>
</tr>
<tr>
<td><code>simenv_put_py(var_name, field)</code></td>
<td>output model results to native SimEnv output file(s)</td>
<td>string <code>var_name</code> (input) declaration of field(...) according to <code>&lt;model&gt;.mdf</code> (input)</td>
<td>name of the variable in <code>&lt;model&gt;.mdf</code> to be output data of variable <code>var_name</code> to be stored as simulation results. Maximum length of field is limited to 12.000 characters.</td>
</tr>
<tr>
<td><code>simenv_slice_py(var_name, idim, ifrom, ito)</code></td>
<td>Currently not available for Python models</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>simenv_end_py()</code></td>
<td>close model coupling interface</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 5.6** Model interface functions for Python models
Due to the special features of Python the coupling interface to SimEnv differs from that for Fortran and C/C++ in Section 5.4. Additionally, Python supports only some data types (check Tab. 5.4). Tab. 5.6 summarizes the model interface functions for a Python model.

- Python model interface functions are declared in the file simenv.py in the SimEnv home directory. To use these functions in a Python model import it by
  
  from simenv import *

  and refer to it for example by simenv_get_py.

- Errors that occur during performance of one of the above functions are directly reported to the log-file <model>.nlog.

In Example 15.5 on page 150 the model world_py.py is described in detail.

### 5.5.1 Standard Shell Scripts for Python Models

<model>.ini

<model>.ini (see Section 7.1 on page 53) is for Python models a mandatory shell script and has to have the same contents for all Python models:

```bash
$SE_HOME/py_model_ini
rc_py_model_ini = $? 

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

For an experiment restart with a Python model (check Section 7.3 on page 55) <model>.ini has to be performed again. To force this specify in <model>.cfg (check Section 10.1 on page 107) for the sub-keyword 'restart_ini' the value "yes".

### 5.6 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 (see Section 7.1 on page 53) is used to wrap the model and optionally to have at disposal corresponding functionality of the SimEnv model interface functions of Tab. 5.5.
<table>
<thead>
<tr>
<th>Dot script name</th>
<th>Command description</th>
<th>Arguments</th>
<th>Argument description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>$SE_HOME/simenv_ini_sh</code></td>
<td>initialize current single run</td>
<td>SE_RUN (output)</td>
<td>operating system environment variable SE_RUN is set to the current run number of the simulation experiment</td>
</tr>
<tr>
<td></td>
<td>Perform always and as the first SimEnv dot script in <code>&lt;model&gt;.run</code> and <code>&lt;model&gt;.rst</code>. Alternatively perform for <code>&lt;model&gt;.run</code> dot script <code>$SE_WS/&lt;model&gt;_sh.inc</code> for semi-automated model interface</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>target_name= '...'</code></td>
<td>get a numerical adjustment in the current single run for the target to be experimented with</td>
<td>script variable <code>target_name</code> (input)</td>
<td>name of the target in <code>&lt;model&gt;.edf</code></td>
</tr>
<tr>
<td><code>target_def_val= ...</code></td>
<td>script variable <code>target_def_val</code> (input)</td>
<td>nominal / default (non-adjusted) target value. If <code>target_name</code> is not defined in <code>&lt;model&gt;.edf</code> then <code>target_adj_val</code> is set to <code>target_def_val</code></td>
<td></td>
</tr>
<tr>
<td></td>
<td>script variable <code>target_name</code> (output)</td>
<td>shell script variable with the same name as the value of <code>target_name</code>. Script variable value is the adjusted target value <code>target_adj_val</code>.</td>
<td></td>
</tr>
<tr>
<td><code>$SE_HOME/simenv_get_run_sh</code></td>
<td>get the run number of the current run as an integer and a character script variable</td>
<td><code>run_char</code> (output)</td>
<td>shell script variable with the current run number with leading zeros</td>
</tr>
<tr>
<td></td>
<td><code>run_int</code> (output)</td>
<td>shell script variable (type integer) with the current run number</td>
<td></td>
</tr>
<tr>
<td><code>$SE_HOME/simenv_get_sh</code></td>
<td>Not available at shell script level</td>
<td>Write a model related simenv_put_sh at the language level using the SimEnv model interface functions from Tab. 5.5 or Tab. 5.6</td>
<td></td>
</tr>
<tr>
<td><code>$SE_HOME/simenv_put_sh</code></td>
<td>Not available at shell script level</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>$SE_HOME/simenv_slice_sh</code></td>
<td>wrap up current single run</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Perform always and as the last SimEnv dot script in <code>&lt;model&gt;.run</code> and <code>&lt;model&gt;.rst</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 5.7  Model interface functions at shell script level**

- For the model interface at the shell script level, i.e., within the shell script `<model>.run` the adjusted experiment targets for the current single run from the whole run ensemble can be made available within `<model>.run` to forward them by any means the modeller is responsible for to the model under investigation.
  
  One common way to forward experiment targets to the model is to place current numerical target values as arguments to the model at the model command line in Unix or Linux. Another way could be to read the targets from a special file in a special file format.

- While for the C/C++/Fortran/Python model interface the names of corresponding targets in the model description file `<model>.edf` and the model source code can differ and are associated by the first argu-
ment of the interface function simenv_put_* (see Fig. 5.1) the names have to coincide for the model inter-
face at the shell script level.
• Directly before performing the dot script $SE_HOME/simenv_get_sh make sure that the shell script vari-
able target_name and target_def_val have been specified. At the end of the dot script simenv_get_sh these variables are set again to empty strings.
• After running the dot script $SE_HOME/simenv_get_sh an experiment target <target_name> from the ex-
periment description file <model>.edf is available in <model>.run as a shell script variable <tar-
get_name> and the adjusted value of the target is available as $<target_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 func-
tions simenv_*_* from the SimEnv model interfaces at language level.
• Tab. 10.10 lists the built-in (pre-defined) shell script variables that are defined and/or used by the dot scripts $SE_HOME/simenv_*_sh and are directly available in <model>.run.
• Please notice:
  To perform a dot script (see the Glossary at the end of this document) it has to be preceded by a dot and a space.

In Example 15.6 on page 151 the model shell script world_sh.run is described in detail.

```
. $SE_HOME/simenv_ini_sh
#
# get adjusted value for the a target p_def, defined in the edf-file
# target_name='p_def'
# target_def_val=2.
. $SE_HOME/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 target p_undef, not defined in edf-file
# target_name='p_undef'
# target_def_val=-999.
. $SE_HOME/simenv_get_sh
#
# now shell script variable p_undef  is available
# value of shell script variable p_undef  is  -999.
#
# ...
.
$SE_HOME/simenv_end_sh
```

Example file: world_sh.run

**Example 5.2**  Addressing target names and values for the model interface at shell script level

### 5.7 Model Interface for GAMS Models

SimEnv allows to interface GAMS models to the experiment shell. A GAMS model for SimEnv can consist of a GAMS main model and GAMS sub-models. Therefore, two additional include-statements have to be inserted into these GAMS model source code files where experiment targets 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 a number of GAMS sub-model that are called directly from the main model. All these GAMS model source code files have to be located in the current workspace. Additional GAMS sub-programs (included files) are not affected bei SimEnv, but one should keep in mind that the GAMS code within SimEnv will be executed in a sub-directory of the current workspace (see below) and so the include statements have to be changed, if the files are addressed in a relative manner (see below).
• The include files are
  `<GAMS_model>_simenv_get.inc`
  `<GAMS_model>_simenv_put.inc`
  where `<GAMS_model>` is the name of a GAMS model file without extension .gms under consideration.
• During experiment preparation the file `<GAMS_model>_simenv_put.inc` and during experiment performance files `<GAMS_model>_simenv_get.inc` are generated automatically to forward GAMS model output to SimEnv data structures and to adjust investigated experiment targets, respectively. These include files correspond with the `simenv_put` and `simenv_get` model interface functions at the language level (see Section 5.4).
• The GAMS include statement `$include <GAMS_model>_simenv_get.inc` has to be placed in the GAMS model file at such a position where all the GAMS variables are declared. Directly before the include statement the target default values have to be assigned to target variables, that are introduced additionally in the model. Directly after the include statement the target variables with the adjusted target values have to be assigned to the model output variables.
• The GAMS include statement `$include <GAMS_model>_simenv_put.inc` has to be placed in the GAMS model file at such a position where all the variables from the model output description file can be output by GAMS put-statements.
• In the course of experiment preparation the GAMS model and all sub-models that are specified in `<model>.gdf` (see below) are transformed automatically. Each GAMS model single run from the run ensemble is performed in a separate sub-directory of the current workspace. Transformed GAMS models and sub-models are copied to this sub-directory and are performed from there. Keep this in mind when specifying in any GAMS model include statements with relative paths.

In Example 15.8 on page 154 the model `gams_model.gms` is described in detail.

Additionally, the following settings are valid:
• An ASCII GAMS description file `<model>.gdf` (see below) has to be supplied to specify the GAMS sub-models and assigned targets and model output variables in detail.
• Maximum dimensionality of any model output variable declared in `<model>.mdf` is 4 for GAMS models.

Note the following information:
• To output the GAMS model status to SimEnv a
  
  ```
  PARAMETER modstat
  has to be declared and the statement
  modstat = `<model_name>`.modelstat
  ```

  has to be incorporated in the GAMS model above the `$include <GAMS_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`.

5.7.1 Standard Shell Scripts for GAMS Models

`<model>.ini`

`<model>.ini` (see Section 7.1 on page 53) is for GAMS models a mandatory shell script and has to have the contents for all GAMS models:

```bash
$SE_HOME/gams_model_ini
rc_gams_model_ini = $?

# additional user-model specific commands can be implemented up from here
if test $rc_gams_model_ini = 0
  then
    fi
  fi
exit $rc_gams_model_ini
```
For an experiment restart with a GAMS model (check Section 7.3 on page 55) <model>.ini has to be performed again. To force this specify in <model>.cfg (check Section 10.1 on page 107) for the sub-keyword 'restart_in' the value "yes".

<model>.run
<model>.run (see Section 7.1 on page 53) has for each GAMS model the same contents:

```bash
#! /bin/sh
. $SE_HOME/simenv_ini_sh
. $SE_HOME/gams_model_run
. $SE_HOME/simenv_end_sh
```

<model>.end
<model>.end (see Section 7.1 on page 53) is for GAMS models a mandatory shell script and has to have the contents for all GAMS models:

```bash
$SE_HOME/gams_model_end

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

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

<model>.edf
While for the C/C++/Fortran/Python model interface the names of corresponding targets 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_* (see Fig. 5.1) the names have to coincide for the GAMS model interface. In the GAMS model code the targets specified in the experiment description file have to be of type PARAMETER and have be defined before the include statement $include simenv_get.inc.

<model>.mdf
Corresponding variables in the model output description file and in the GAMS model source code must have same names. The variable has 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 GAMS model output is restricted to 4.

With respect to Example 15.8 the model output description file could look like

```plaintext
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 5.3 Model output description file for a GAMS model

5.7.2 GAMS Description File <model>.gdf

The ASCII GAMS description file <model>.gdf is intended to create a block of lines for each GAMS sub-model with a simenv_get.inc file and/or a simenv_put.inc file. The block 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 targets from the target description file have to be used in this file again.

<model>.gdf is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 5.3.

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

**Tab. 5.8**  
Elements of a GAMS description file <model>.gdf

To Tab. 5.8 the following additional rules and explanations apply:
- For the description of **line type** check Tab. 11.4 on page 121.
- Each target and each model output variable as declared in <model>.edf and <model>.mdf respectively has to be used in the value-field of <model>.gdf exactly one time.
- To each GAMS model <model_name> an arbitrary number of targets and model output variables can be assigned to by the corresponding sub-keyword 'get' and/or 'put'. To each sub-model ('type' = "sub") at least one 'get' or one 'put' sub-keyword must be assigned to. The main model ('type' = "main") can be configured without any sub-keyword 'get' and 'put'. This is useful when the main model simply calls sub-models.
- Each model <model_name> in <model>.gdf with at least one sub-keyword 'get' has to have an $include <model_name>_simenv_get.inc statement in the corresponding GAMS model file <model_name>.gms
- Each model <model_name> in <model>.gdf with at least one sub-keyword 'put' has to have an $include <model_name>_simenv_put.inc statement in the corresponding GAMS model file <model_name>.gms
- There has to be exactly one main GAMS model, identified by the sub-keyword 'type' value "main". All other models have to be of sub-keyword type value "sub".
- The value-field for the sub-keyword 'put' is adapted to GAMS syntax to output GAMS model output variables. Afterwards this output is used to generate the appropriate SimEnv output.
- The sub-keyword 'time_limit' enables limitation of each GAMS model single run in the run ensemble to a maximum CPU-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 CPU time consumption. If the sub-keyword is not used in the gdf-file CPU-time consumption per single run is unlimited.

With respect to Example 15.8 the GAMS description file could look like

```plaintext
with respect to Example 15.8 the GAMS description file could look like

general descr GAMS model output description
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 5.4**  
GAMS description file <model>.gdf

Example file: gams_model.gdf
If the model gams_model from the above Example 5.5 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)

```gams
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 5.5**  
*GAMS description file for coupled GAMS models*

### 5.7.3 Files Created during GAMS Model Performance

Additionally to the files listed in Tab. 10.7, 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 from the experiment is performed individually in a directory `run<run_char>` of the current workspace. Each directory is generated automatically before performing the corresponding single run and removed after performace 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.

Unlike the other interface implementations GAMS main model terminal output for each single run is redirected to the log-file `<model>.nlog` in the directory `run<run_char>`. The modeler is responsible for re-direction of the terminal output from sub-models and from solvers. It is recommended to call all GAMS sub-models with the GAMS option string

```
ll=0 lo=2 lf=<model>.nlog dp=0
```

(see Example 15.8) which is also applied for the main model. With the options sub-keyword ‘options’ additional options can be specified in `<model>.cfg` for the main model.
5.8 Semi-Automated Model Interface

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

- **Initialization:** `simenv_ini_*` and `simenv_get_run_*`
- **Target adjustments:** `simenv_get_*`
- **Model output:** `simenv_slice_*` and `simenv_put`
- **End:** `simenv_end_*`

Often, initialization and target adjustments can be lumped together in a source code sequence where the target adjustment part has to be updated when new targets are defined in an experiment description file and have to be mapped to model internal targets 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 target adjustments are generated automatically during experiment preparation as sequences of source code based on the current experiment description file (and consequently the current experiment targets) for all supported model source code languages but GAMS.

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 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 target 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 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` (see Section 10.1).

The Fortran/C/C++/Python model interfaces offer to use different names of corresponding targets 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_*` (see Fig. 5.1). **When using the semi-automated model interface the SimEnv target names and the corresponding source code entity names have to be coincided.**

Automatically generated source code sequences are stored in files `<model>_<f|c|py|sh>.inc` in the current workspace `$SE_WS`. When using targets `t1` and `t2` in the experiment description file `<model>.edf` then the source code sequences have the following contents:

For Fortran:

```plaintext
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 ( 't1' , 0. , t1 )
simenv_sts = simenv_get_f ( 't2' , 0. , t2 )
```

For C/C++:

```plaintext
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 ( "t1" , &simenv_zero , &t1 )
simenv_sts = simenv_get_c ( "t2" , &simenv_zero , &t2 )
```
for Python:

```python
from simenv import *

simenv_ini_py()

t1 = float(simenv_get_py('t1', 0.))
t2 = float(simenv_get_py('t2', 0.))
```

for the model interface at shell script level:

```bash
$SE_HOME/simenv_ini_sh
$SE_HOME/simenv_get_run_sh
target_name='t1'
target_def_val=0.
$SE_HOME/simenv_get_sh
target_name='t2'
target_def_val=0.
$SE_HOME/simenv_get_sh
```

The sequence of targets in the code sequences corresponds to the sequence of targets in the experiment description file `<model>.edf`. For more than two targets the code sequences are enlarged accordingly.

For the Fortran/C/C++ model interface:
- the variables `simenv_sts`, `simenv_run_int`, `simenv_run_char`, and `simenv_zero` are defined in the model source code include file `simenv_mod_[f|c].inc` (see Section 10.3).
- model link files `<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 (for service `simenv.run`, but not for `simenv.rst`).

The source code sequences are included in the model source code by

```bash
for Fortran: include '<model>_f.inc'
for C/C++: #include "<model>_c.inc"
for Python: from simenv import *
```

for the model interface at shell script level:

```bash
$SE_WS/<model>_sh.inc
```

Examples can be found in Example 15.2 and Example 15.7.

### 5.9 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` (see Section 10.1) by the sub-keyword ‘structure’. Lumped (standard) models are normally represented by one stand-alone executable. A distributed model in SimEnv consists from a web of stand-alone sub-models, i.e., the model dynamics are computed by performing a set of stand-alone sub-models that normally interact with each other and exchange information. For a parallel model each single run of an experiment needs a set of assign processors.

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

For distributed models each of the sub-models can use SimEnv model interface functionality, i.e., `simenv_get_*`, `simenv_get_run_*`, `simenv_put_*`, or `simenv_slice_*`. In each sub-model with SimEnv model interface functionality `simenv_ini_*` and `simenv_end_*` calls have to be incorporated in. Sub-models can be implemented in different programming languages. Additionally, the corresponding SimEnv model interface functionality at shell script level (simenv_*_sh dot scripts) can be applied. As usual, the overall model is wrapped into a shell script `<model>.run` (see 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 targets 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 target 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 the model to the load leveler LoadL by using the llsubmit command. Start an experiment
from a login-node of the compute cluster and run the experiment at the login machine. SimEnv submits from
the login machine all single runs to LoadL and directly finishes afterwards. The parallel operating environ-
ment POE and the load leveler LoadL then take responsibility to perform the single model runs.
For the parallel modus the temporary SimEnv files <model>_\_*\_bin and simenv_\_*\_tmp are not deleted at ex-
periment end, i.e. after all single model runs are submitted. These files can be removed manually after finish-
ing the last single run by POE. Check the LoadL services for the end of the last parallel single model run.
To support bookkeeping of SimEnv applications on PIK’s parallel cluster machine please 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
```

Example 5.6  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.

5.10 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
  `$SE_HOME/libsimenvdummy.a`
  instead of
  `$SE_HOME/libsimenv.a`
  For this library
  - SimEnv model interface function values (return codes) are 0
  - simenv_get\_*\_* forwards target_def_val to target_adj_val
  - simenv_get_run\_*\_* returns integer run number 0 and character run string ‘ ’ (six blanks).
For Python models:
Replace in the model source code
    from simenv import *
by
    from simenvdummy import *

For this module
• SimEnv model interface function values (return codes) are 0
• simenv_get_py forwards target_def_val to target_adj_val
• simenv_get_run_py returns run 000000.

For GAMS models:
Handle in the model source code the include statements
    $include <GAMS_model>_simenv_get.inc
and
    $include <GAMS_model>_simenv_put.inc
as 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 information from the model and numerical information (see Chapter 4). An equipped experiment type is represented by an experiment description file `<model>.edf`.

 `<model>.edf` is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and value as in Tab. 6.1.

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>&lt;nil&gt;</td>
<td>descr</td>
<td>o</td>
<td>any</td>
<td>&lt;string&gt;</td>
<td>experiment description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
<td>m</td>
<td>1</td>
<td>[behaviour</td>
<td>monte carlo</td>
</tr>
<tr>
<td>target</td>
<td>&lt;target_name&gt;</td>
<td>descr</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>target description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unit</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>target unit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>type</td>
<td>m</td>
<td>1</td>
<td>see Tab. 6.2</td>
<td>adjustment type: specifies how to modify the sampled values by the target default value in simenv_get_ *</td>
</tr>
<tr>
<td></td>
<td></td>
<td>default</td>
<td>m</td>
<td>1</td>
<td>&lt;real_val&gt;</td>
<td>target default value &lt;target_def_val&gt;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>adjusts</td>
<td>c3</td>
<td>1</td>
<td>&lt;experiment specific&gt;</td>
<td>specifies how to sample the target to get adjustment values &lt;adj_val&gt;</td>
</tr>
<tr>
<td>specific</td>
<td>&lt;nil&gt;</td>
<td>&lt;experiment specific&gt;</td>
<td>m</td>
<td>&lt;experiment specific&gt;</td>
<td>&lt;experiment specific&gt;</td>
<td>experiment specific information</td>
</tr>
</tbody>
</table>

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

To Tab. 6.1 the following additional rules and explanations apply:
- For the description of line type check Tab. 11.4 on page 121.
- A target name is the symbolic parameter / driver / initial value / boundary value name, corresponding to targets of the investigated model. Correspondence is achieved by applying the SimEnv model interface function `simenv_get_ *` in the model.
- Target names must differ from model output variables and coordinate names in the model output description file (see Section 5.1) and from built-in and user-defined operator names for experiment post-processing (see Section 8.5.4).
• To derive the adjusted value the default value as specified in <model>.edf and not the default value from the model code is used in the model interface function simenv_get_*.
• For adjustment types multiply and relative default <real_val> = 0. is forbidden.
• All experiment specific information is explained in the appropriate sections.
• Specify at least one experiment target.
• When preparing an experiment an experiment input file <model>.edf_adj is generated with the sampled adjustment values <adj_val> according to the information in the sub-keyword ‘adjusts’. These values are applied within the interface function simenv_get_* to the default values of the targets according to the specified adjustment type (see Tab. 6.2 below) before finally influencing the dynamics of the model. The sequence of elements (columns) of each record of <model>.edf_adj corresponds with the sequence of targets in the target name space (see Section 11.1 on page 119), the sequence of records corresponds with the sequence of single model runs of the experiment.
• For each experiment a single model run with run number <run_int> = 0 and <run_char> = 000000 is generated automatically as the nominal run of the model without adjustments. This run does not have an assigned record in <model>.edf_adj.

<table>
<thead>
<tr>
<th>Adjustment type</th>
<th>Meaning:</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
<td>To derive the final adjusted value &lt;target_adj_val&gt; to use in the model from the sampled adjustment value &lt;adj_val&gt; (from &lt;model&gt;.edf_adj) and the target default value &lt;target_def_val&gt; (as defined in &lt;model&gt;.edf) within the SimEnv model interface function simenv_get_* the sampled adjustment value is … set … set to the adjusted target value: &lt;target_adj_val&gt; = &lt;adj_val&gt;</td>
</tr>
<tr>
<td>add</td>
<td>… added to the target default value: &lt;target_adj_val&gt; = &lt;adj_val&gt; + &lt;target_def_val&gt;</td>
</tr>
<tr>
<td>multiply</td>
<td>… multiplied by the target default value: &lt;target_adj_val&gt; = &lt;adj_val&gt; * &lt;target_def_val&gt;</td>
</tr>
<tr>
<td>relative</td>
<td>… increased by 1 and afterwards multiplied by the target default value: &lt;target_adj_val&gt; = (1. + &lt;adj_val&gt;) * &lt;target_def_val&gt;</td>
</tr>
</tbody>
</table>

Tab. 6.2 Adjustment types in experiment preparation

6.2 Behavioural Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for behavioural analysis as follows:

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>&lt;target_name&gt;</td>
<td>adjusts</td>
<td>a</td>
<td>1</td>
<td>&lt;val_list&gt;</td>
<td>value list of target value adjustments &lt;adj_val&gt; (for syntax see Tab. 11.6)</td>
</tr>
<tr>
<td>specific</td>
<td>&lt;nil&gt;</td>
<td>comb</td>
<td>m</td>
<td>≥ 1</td>
<td>[ default</td>
<td>&lt;combination&gt;</td>
</tr>
</tbody>
</table>

Tab. 6.3 Experiment specific elements of an edf-file for behavioural analysis

To Tab. 6.3 the following additional rules and explanations apply:
• For the description of line type check Tab. 11.4 on page 121.
For sub-keyword ‘comb’ the following rule holds:

- Value = [ default | <combination> ] for available sub-keyword ‘adjusts’
- Value = [ file {<directory>/}<file_name> ] for unavailable sub-keyword ‘adjusts’

Values of a value list have to be unique for available sub-keyword ‘adjusts’ and each target. Assigned values from file {<directory>/}<file_name> can be multiple defined for each target.

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

6.2.1 The Combination

- The combination <combination> defines the way in which the space spanned by the experiment targets will be inspected by SimEnv: This is done by applying operators "*" and "," to all stated experiment targets.
  - The operator "*" combines adjustments of different targets and so their resulting values combinatorially ("for all mesh points in a grid").
  - Compare with experiment description file (a) from Example 6.1 below.
  - The operator "," combines adjustments of different targets and so their resulting values parallel ("on the diagonal").
    - For the operator "," the targets must have the same number of adjustments.
    - Compare with experiment description file (b) from Example 6.1 below.
  - The operator "," has a higher priority than the operator "*". Parentheses are not allowed:
    - For example, p1 * p2 , p3 * p4 always combines p2 and p3 in parallel and this combinatorially with p1 and p4. A parallel combination of p1 * p2 with p3 * p4 by (p1 * p2) , (p3 * p4) is not possible.
    - Compare with experiment description file (c) from Example 6.1 below.
  - In <combination> each target has to be used exactly once.

- By the default combination default all experiment targets are combined combinatorially.
  - comb default of the experiment description file (a) from Example 6.1 below is equivalent to comb p1 * p2.
  - Specification of file is only allowed for unused sub-keywords ‘adjusts’ all over the edf-file.
  - The adjustments are read from the adjustment data file {<directory>/}<file_name>.
  - All targets are assumed to be combined in parallel. Each record of the data file represents one simulation run. The sequence of the adjustments (sequence of columns) in each record corresponds with the sequence of the targets in the target name space (see Section 11.1 on page 119).
  - Syntax rules for value lists on page 119 hold.
  - Identical adjustments for a target are allowed.
  - During experiment post-processing restricted capabilities for the operator behav apply for this experiment layout.
  - Compare with experiment description file (d) from Example 6.1 below. Combination is implicitly as comb p1 , p2. Experiment description files (b) and (d) in Example 6.1 below describe the same experiment.
  - To continue a combination <combination> at a following comb-line end the current comb-line by one of the operators "*" or ",".
6.2.2 Example

The first three experiment description files (a) to (c) represent an experiment description according to Fig. 4.3 (a) to (c) on page 14. Results in adjusted target values

<table>
<thead>
<tr>
<th>(a)</th>
<th>general descr</th>
<th>Experiment description for the examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>general descr</td>
<td>in the SimEnv User Guide (Fig. 4.3 (a))</td>
</tr>
<tr>
<td></td>
<td>general type</td>
<td>behaviour</td>
</tr>
<tr>
<td></td>
<td>target p1 descr</td>
<td>parameter p1</td>
</tr>
<tr>
<td></td>
<td>target p1 unit</td>
<td>without</td>
</tr>
<tr>
<td></td>
<td>target p1 type</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>target p1 default</td>
<td>1.</td>
</tr>
<tr>
<td></td>
<td>target p1 adjusts</td>
<td>list 1, 3, 7, 8</td>
</tr>
<tr>
<td></td>
<td>target p2 descr</td>
<td>parameter p2</td>
</tr>
<tr>
<td></td>
<td>target p2 unit</td>
<td>without</td>
</tr>
<tr>
<td></td>
<td>target p2 type</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>target p2 default</td>
<td>2.</td>
</tr>
<tr>
<td></td>
<td>target p2 adjusts</td>
<td>list 1, 2, 3, 4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(b)</th>
<th>general descr</th>
<th>Fig. 4.3 (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>general type</td>
<td>behaviour</td>
</tr>
<tr>
<td></td>
<td>target p1 type</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>target p1 default</td>
<td>1.</td>
</tr>
<tr>
<td></td>
<td>target p1 adjusts</td>
<td>list 1, 3, 7, 8</td>
</tr>
<tr>
<td></td>
<td>target p2 type</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>target p2 default</td>
<td>2.</td>
</tr>
<tr>
<td></td>
<td>target p2 adjusts</td>
<td>equidist_end 1(0.5)2.5</td>
</tr>
<tr>
<td></td>
<td>specific comb</td>
<td>default</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(c)</th>
<th>general descr</th>
<th>Fig. 4.3 (c)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>general type</td>
<td>behaviour</td>
</tr>
<tr>
<td></td>
<td>target p1 type</td>
<td>set</td>
</tr>
<tr>
<td></td>
<td>target p1 default</td>
<td>1.</td>
</tr>
<tr>
<td></td>
<td>target p1 adjusts</td>
<td>list 1, 3, 7, 8</td>
</tr>
<tr>
<td></td>
<td>target p2 type</td>
<td>set</td>
</tr>
<tr>
<td></td>
<td>target p2 default</td>
<td>2.</td>
</tr>
<tr>
<td></td>
<td>target p2 adjusts</td>
<td>equidist_end 1(1)4</td>
</tr>
<tr>
<td></td>
<td>target p3 type</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>target p3 default</td>
<td>3.</td>
</tr>
<tr>
<td></td>
<td>target p3 adjusts</td>
<td>list 1.1, 1.5, 2.4</td>
</tr>
<tr>
<td></td>
<td>specific comb</td>
<td>p2,p1*p3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>(d)</th>
<th>general descr</th>
<th>Fig. 4.3 (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>general type</td>
<td>behaviour</td>
</tr>
<tr>
<td></td>
<td>target p1 type</td>
<td>multiply</td>
</tr>
<tr>
<td></td>
<td>target p1 default</td>
<td>1.</td>
</tr>
<tr>
<td></td>
<td>target p2 type</td>
<td>add</td>
</tr>
<tr>
<td></td>
<td>target p2 default</td>
<td>2.</td>
</tr>
<tr>
<td></td>
<td>specific comb</td>
<td>file world.dat_d:</td>
</tr>
</tbody>
</table>

Example files: world.edf_a to world.edf_d

Example 6.1 Experiment description file <model>.edf for behavioural analysis
6.3 Monte Carlo Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for Monte Carlo analysis as follows:

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>&lt;target_name&gt;</td>
<td>adjusts</td>
<td>m</td>
<td>1</td>
<td>[ &lt;distribution&gt;</td>
<td>file {&lt;directory&gt;/}</td>
</tr>
<tr>
<td>sample</td>
<td>c4</td>
<td></td>
<td>1</td>
<td></td>
<td>[ random</td>
<td>latin hypercube ]</td>
</tr>
<tr>
<td>specific</td>
<td>&lt;nil&gt;</td>
<td>runs</td>
<td>m</td>
<td>1</td>
<td>&lt;int_val&gt;</td>
<td>number of runs &gt; 10 to be performed for the experiment</td>
</tr>
</tbody>
</table>

Tab. 6.4 Experiment specific elements of an edf-file for Monte Carlo analysis

To Tab. 6.4 the following additional rules and explanations apply:
- For the description of line type check Tab. 11.4 on page 121.
- For <distribution> = <distr_shortcut> ( <distr_param_1> {, <distr_param_2> } ) check Tab. 6.5.
- For implicitly specified distributions according to Tab. 6.5 sample values <adj_val> is generated from the distribution with the assigned distribution parameters.
- For an explicitly ASCII file <file_name> sample values of any distribution are taken directly from this file. For syntax rules for files check Section 11.1. Each record of the ASCII file can hold only one sample value. Sample size has to be identical to <nr_of_runs> from the keyword ‘specific’.
- In random sampling, there is no assurance that sampling points will cover all regions of the selected distribution. With Latin hypercube sampling LHS this shortcoming is reduced: The sampling range of the target is divided into <nr_of_runs> intervals of equal probability according to the selected distribution and from each interval exactly one sampling point is drawn. For more information on LHS check Fig. 6.1 below and see Imam & Helton (1998) and Helton & Davis (2000).
- The number of runs must be greater than 10.

6.3.1 Distribution Functions and their Parameters

<table>
<thead>
<tr>
<th>Distribution function</th>
<th>distr_shortcut</th>
<th>distr_param_1</th>
<th>distr_param_2</th>
<th>Restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>uniform</td>
<td>U</td>
<td>lower boundary</td>
<td>upper boundary</td>
<td>lower boundary &lt; upper boundary</td>
</tr>
<tr>
<td>normal</td>
<td>N</td>
<td>mean value</td>
<td>variance</td>
<td>variance &gt; 0</td>
</tr>
<tr>
<td>lognormal</td>
<td>L</td>
<td>mean value of a normally distributed target</td>
<td>variance of a normally distributed target</td>
<td>variance &gt; 0</td>
</tr>
<tr>
<td>exponential</td>
<td>E</td>
<td>mean value</td>
<td>---</td>
<td>mean value &gt; 0</td>
</tr>
</tbody>
</table>

Tab. 6.5 Probability density functions and their parameters

For more information on the distribution functions see Section 4.3 and Tab. 4.2.
Be careful when specifying for a Monte Carlo analysis an adjustment type (see Tab. 6.2) that differs from 'set'. For adjustment type 'add' normally the mean value of the sample will be shifted by the specified target default value <target_def_val>. For adjustment types 'multiply' and 'relative' the specified distribution will be adulterated normally by the target default value <target_def_val>.

Fig. 6.1  Monte Carlo analysis: Latin hypercube sampling

6.3.2 Example

<table>
<thead>
<tr>
<th>general descr</th>
<th>Experiment description for the examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>general descr</td>
<td>in the SimEnv User Guide</td>
</tr>
<tr>
<td>general type</td>
<td>Monte Carlo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>target p2 descr</th>
<th>parameter p1</th>
</tr>
</thead>
<tbody>
<tr>
<td>target p2 unit</td>
<td>without</td>
</tr>
<tr>
<td>target p2 type</td>
<td>multiply</td>
</tr>
<tr>
<td>target p2 default</td>
<td>2.</td>
</tr>
<tr>
<td>target p2 sample</td>
<td>latin hypercube</td>
</tr>
<tr>
<td>target p2 adjusts</td>
<td>distr U(0.5,1.5)</td>
</tr>
</tbody>
</table>

p2 is sampled from a uniform distribution between 0.5 and 1.5. In simenv_get_*, each value is multiplied by 2.

<table>
<thead>
<tr>
<th>target p1 type</th>
<th>add</th>
</tr>
</thead>
<tbody>
<tr>
<td>target p1 default</td>
<td>1.</td>
</tr>
<tr>
<td>target p1 sample</td>
<td>random</td>
</tr>
<tr>
<td>target p1 adjusts</td>
<td>distr N(0,0.4)</td>
</tr>
</tbody>
</table>

p1 is sampled from a normal distribution with mean = 1 and variance = 0.4. In simenv_get_*, each value is increased by 1.
target p3 type set
target p3 default 3.
target p3 adjusts file world.dat_e sample for p3 is read from file
world.dat_e

specific runs 250

Example file: world.edf_e

**Example 6.2** Experiment description file `<model>.edf` for Monte Carlo analysis

### 6.4 Local Sensitivity Analysis

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for local sensitivity analysis as follows:

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td><code>&lt;target_name&gt;</code></td>
<td>adjusts</td>
<td>f</td>
<td>0</td>
<td></td>
<td>sub-keyword is forbidden for this experiment type</td>
</tr>
<tr>
<td>specific</td>
<td><code>&lt;nil&gt;</code></td>
<td>incrs</td>
<td>m</td>
<td>1</td>
<td><code>&lt;val_list&gt;</code></td>
<td>Increment values that form a sample of target value adjustments <code>&lt;adj_val&gt;</code> Increments <code>&lt;adj_val&gt;</code> &gt; 0. <code>&lt;adj_val&gt;</code> in <code>&lt;val_list&gt;</code> has to be ordered in a strictly monotonic increasing manner. (for syntax see Tab. 11.6)</td>
</tr>
</tbody>
</table>

**Tab. 6.6** Experiment specific elements of an edf-file for local sensitivity analysis

To Tab. 6.6 the following additional rules and explanations apply:
- For the description of line type check Tab. 11.4 on page 121.
- For experiment type local sensitivity analysis only the adjustment types add and relative are allowed.
- Values from the value list must be positive and unique.

### 6.4.1 Sensitivity Functions and Run Sequence

As an example, the absolute sensitivity function (see Tab. 4.3 on page 17) is as follows:

\[
\text{sens}_\text{abs}(\text{<target}_\text{def_val}>, \pm <\text{adj}_\text{val}>) = \frac{\text{z}(\text{<target}_\text{def_val}) \pm <\text{adj}_\text{val}>}{\pm <\text{adj}_\text{val}>} - \text{z}(\text{<target}_\text{def_val}) \\
\text{sens}_\text{abs}(\text{<target}_\text{def_val}>, \pm <\text{adj}_\text{val}>) = \frac{\text{z}(\text{<target}_\text{def_val}) \pm <\text{adj}_\text{val}>}{\pm <\text{target}_\text{def_val}>} - \text{z}(\text{<target}_\text{def_val}) \\
\text{sens}_\text{abs}(\text{<target}_\text{def_val}>, \pm <\text{adj}_\text{val}>) = \frac{\text{z}(\text{<target}_\text{def_val}) \pm <\text{adj}_\text{val}>}{\pm <\text{target}_\text{def_val}>} \times <\text{adj}_\text{val}>
\]

for adjustment type Add

for adjustment type Relative
The sequence of the single simulation runs is determined in the following manner:

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

6.4.2 Example

<table>
<thead>
<tr>
<th>(f)</th>
<th>general descr</th>
<th>Experiment description for the examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>general descr</td>
<td>in the SimEnv User Guide</td>
</tr>
<tr>
<td></td>
<td>general type</td>
<td>local sensitivity</td>
</tr>
<tr>
<td>target p1 descr</td>
<td>parameter p1</td>
<td></td>
</tr>
<tr>
<td>target p1 unit</td>
<td>without</td>
<td></td>
</tr>
<tr>
<td>target p1 type</td>
<td>add</td>
<td></td>
</tr>
<tr>
<td>target p1 default</td>
<td>1.</td>
<td></td>
</tr>
<tr>
<td>target p2 type</td>
<td>relative</td>
<td></td>
</tr>
<tr>
<td>target p2 default</td>
<td>2.</td>
<td></td>
</tr>
<tr>
<td>target p3 type</td>
<td>relative</td>
<td></td>
</tr>
<tr>
<td>target p3 default</td>
<td>3.</td>
<td></td>
</tr>
<tr>
<td>specific incrs</td>
<td>list 0.001,0.01,0.05,0.1</td>
<td></td>
</tr>
</tbody>
</table>

Example file: world.edf

**Example 6.3**  Experiment description file `<model>.edf` for local sensitivity analysis
6.5 Optimization

The experiment specific information for experiment description files in Tab. 6.1 on page 43 is defined for local sensitivity analysis as follows:

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>target</td>
<td>&lt;target_name&gt;</td>
<td>adjusts</td>
<td>m</td>
<td>1</td>
<td>&lt;real_val1&gt;:&lt;real_val2&gt;</td>
<td>lower bound &lt;real_val1&gt; and upper bound &lt;real_val2&gt; to define the target range where the cost function is to be minimized on. &lt;real_val1&gt; ≤ &lt;real_val2&gt; Target adjustment values &lt;adj_val&gt; are sampled in this target range.</td>
</tr>
<tr>
<td>specific</td>
<td>&lt;nil&gt;</td>
<td>cost_fct</td>
<td>m</td>
<td>≥ 1</td>
<td>&lt;result&gt;</td>
<td>cost function to minimize. A 0-dimensional result formed according to the rules of the SimEnv post-processor. Do not apply multi-run operators. Cost function definition can be arranged at a series of cost_fct-lines in analogy to the rules for result expressions (see Section 8.1.1).</td>
</tr>
<tr>
<td>max_runs</td>
<td></td>
<td></td>
<td>m</td>
<td>1</td>
<td>&lt;int_val&gt;</td>
<td>number of single runs to end the experiment without checking the other optimization method related stopping criteria.</td>
</tr>
</tbody>
</table>

Tab. 6.7 Experiment specific elements of an edf-file for an optimization experiment

To Tab. 6.7 the following additional rules and explanations apply:
- For the description of line type check Tab. 11.4 on page 121.

6.5.1 Special Features in Optimization

- This is the only experiment type where the adjustments for the targets 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>.edf_adj is created during experiment preparation. The records belonging to the performed single runs are written during experiment performance.
- In parallel to the file model>.edf_adj an ASCII file <model>.edf_cf 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_opt from the SimEnv home directory to <model>.opt_opt in the current workspace and edit this file. During the experiment the edited file is used instead of the file with the default constellation in the SimEnv home directory. The description of the options and parameters can be found in Ingber (2004).
- Optimization experiments can not be restarted by the SimEnv service simenv.rst.
- The values for the sub-keywords ‘begin_run’ and ‘end_run’ in the configuration file <model>.cfg are ignored for an optimization experiment. The experiment always starts with run number 0 and ends if one of the criteria in the file [ <model> | simenv ].opt_opt (see above) is fulfilled or the explicitly stated end run number from the sub-keyword ‘max_runs’ <model>.edf is reached.
As the results of the optimization experiment the optimization return code, the optimal targets, the corresponding value of the cost function and the number of the corresponding single run are documented at the end of the model interface log-file `<model>.mlog`.

A protocol from the optimization procedure is delivered by SimEnv in the ASCII file `<model>.olog`.

### 6.5.2 Example

<table>
<thead>
<tr>
<th>(g)</th>
<th>general</th>
<th>descr</th>
<th>Experiment description for the examples</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>general</td>
<td>descr</td>
<td>in the SimEnv User Guide</td>
</tr>
<tr>
<td></td>
<td>general</td>
<td>type</td>
<td>optimization</td>
</tr>
<tr>
<td>target</td>
<td>p1</td>
<td>descr</td>
<td>parameter p1</td>
</tr>
<tr>
<td>target</td>
<td>p1</td>
<td>unit</td>
<td>without</td>
</tr>
<tr>
<td>target</td>
<td>p1</td>
<td>type</td>
<td>set</td>
</tr>
<tr>
<td>target</td>
<td>p1</td>
<td>default</td>
<td>1.</td>
</tr>
<tr>
<td>target</td>
<td>p1</td>
<td>adjusts</td>
<td>-12:12</td>
</tr>
<tr>
<td>target</td>
<td>p2</td>
<td>type</td>
<td>set</td>
</tr>
<tr>
<td>target</td>
<td>p2</td>
<td>default</td>
<td>2.</td>
</tr>
<tr>
<td>target</td>
<td>p2</td>
<td>adjusts</td>
<td>1:10</td>
</tr>
<tr>
<td>target</td>
<td>p3</td>
<td>type</td>
<td>set</td>
</tr>
<tr>
<td>target</td>
<td>p3</td>
<td>default</td>
<td>3.</td>
</tr>
<tr>
<td>target</td>
<td>p3</td>
<td>adjusts</td>
<td>-12:12</td>
</tr>
<tr>
<td>target</td>
<td>p4</td>
<td>type</td>
<td>set</td>
</tr>
<tr>
<td>target</td>
<td>p4</td>
<td>default</td>
<td>4.</td>
</tr>
<tr>
<td>target</td>
<td>p4</td>
<td>adjusts</td>
<td>1:10</td>
</tr>
</tbody>
</table>

**Specific**

- cost_fct: `-sum(bios)` maximize sum(bios) over land masses
- max_runs: 700

*Example file: world.edf_g*

**Example 6.4** Experiment description file `<model>.edf` for an optimization experiment
7 Experiment Performance

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

7.1 General Approach

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

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

For successive performance of experiment slices and/or after abnormal experiment interrupt experiments can be re-started. 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 output data structure is appended to the output data that already exists for this experiment.

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

- The model variables to be output during experiment performance are declared in the model output description file <model>.mdf
- The type and the targets of the experiment to be performed are declared in the experiment description file <model>.edf
- Mapping between experiment targets and targets in the model source code is achieved by application of the generic SimEnv model interface function simenv_get_* in the model code or at shell script level.
- Output of model variables declared in <model>.mdf into SimEnv structures is achieved by the application of the generic SimEnv model interface function simenv_put_* (and simenv_slice_*) in the model source code.
- Model output from run number <run> is stored in the file <model>.out<run_char>.[ nc | ieee ] if the sum over all model output variables of a single run is less than the appropriate value specified in <model>cfg. Otherwise, model output from the complete experiment is stored in <model>.outall.[ nc | ieee ].
- For all experiment types a run number 0 with the default values of all experiment targets will is declared additionally to the runs declared in the experiment description file <model>.edf.
- During experiment performance a model interface log-file <model>.mlog is written where the adjustments of experiment target values are stored. 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.
- Do not start / restart / submit another experiment from a workspace where an experiment is still running.
- After the experiment has been finished an e-mail is send on demand (check Section 10.1) to the address as specified in <model>.cfg.
- The status of any running experiment can be acquired by the SimEnv service simenv.sts. For more information check Tab. 10.3.
- For more information check Section 5.1 and 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/simenv_ini_sh` is performed always and as the first SimEnv dot script
  - `./ $SE_HOME/simenv_end_sh` is performed always and as the last SimEnv dot script (see Tab. 5.7 on page 32 and Example 7.1 below).
- To cancel the whole experiment after the performance of the current single run due to any condition of this run make sure a file `$SE_WS/<model>_$run_char.err` exists as an indicator to stop. Create this file in the model or in <model>.run. For the latter
  - Perform `./ $SE_HOME/simenv_get_run_sh` to get the current run number `<run_int>` and `<run_char>` (see Tab. 5.7 on page 32 and Example 7.1 below).
  - Touch the file `$SE_WS/<model>_$run_char.err`.
- Terminal output from <model>.run is redirected to the log-file <model>.nlog.
- For GAMS models <model>.run has a pre-defined structure. Check Section 5.7.1 for more information.

- 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 (see Section 7.3 below).
  - In <model>.ini additional settings / checks can be performed. For return codes unless 0 from <model>.ini the experiment will not be started.
  - Terminal output from <model>.ini is also re-directed to the log-file <model>.nlog.
  - For Python and GAMS models <model>.ini is a mandatory shell script with standardized contents. Check Sections 5.5.1 and 5.7.1 for more information.

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

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

---

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

```bash
#!/ /bin/sh

# perform always and as the first $SE_HOME/simenv_*_sh dot script:
./ $SE_HOME/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.
```
```bash
if test $? -ne 0
then
  . $SE_HOME/simenv_get_run_sh
  touch $SE_WS/world_f_$run_char.err
fi

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

```bash
# 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 | sh ].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:

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

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

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

### 7.3 Experiment Parallelization

According to the general SimEnv approach how to use a model the single runs of an experiment are independent from each other. The only exception is the experiment type optimization where the adjustments for the current single run are determined on the outcomes of previous single runs. Keeping this in mind, 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 the single model run the model output data and related information
SimEnv supports distribution of single runs from an experiment for IBM’s cluster architectures. These clusters are managed by the parallel operating environment POE and the load leveler LoadL. The processors of a compute cluster are assigned to job classes 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 processors of a job class (dis - distributed strategy)
  The single runs are submitted to the job class as single jobs in a way that all available processors of the class can be used. Due to controlling the submit process dynamically, the job class will not be overloaded by the single run jobs of the experiment but the submit process will wait on demand. The submit process itself is started in the background.
  The experiment performance will start when a processor of this job class is free.
  Use this strategy for best utilization of all job class processors.

- … on pre-allocated processors of a class (par - parallel strategy)
  A number of processors are assigned to the experiment during experiment preparation and one parallel job is submitted to the job class. During the experiment one communication processor is responsible for experiment management while the other processors serve as simulation processors for the single runs.
  The experiment performance will start when the assigned number of processors are free in this class.
  Use this strategy to make sure to run an experiment in a certain time.

- … sequentially on one processor of a class (seq - sequential strategy)
  One processor of the job class is assigned to the whole experiment and the experiment is performed single run by single run on this processor as a SimEnv experiment performance on the login machine.
  The experiment performance will start when one processor of this job class is free.
  Use this strategy when the other two strategies are impossible (e.g., for an optimization experiment) and you want to use the hardware resources of the compute cluster.

For an experiment performance controlled by the parallel operating environment POE and the load leveler LoadL make sure that the environment variables SE_HOME is set in the file $HOME/.profile correctly.

After the experiment is submitted to the load leveler the current login session can be closed.

Default job control files are supplied by SimEnv to ensure communication with POE and the load leveler. 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 at a parallel or sequential job class.

If necessary, copy the ASCII job control files simenv.jcf_[dis | par | seq] from the SimEnv home directory to <model>.jcf_[dis | par | seq] in the current workspace, modify the file(s) according to the needs of the experiment one want to perform and / or the machine one want to use and start afterwards simenv.run and/or simenv.rst. If available in the current workspace, these modified job control files are used instead of the original files in the SimEnv home directory.

7.4 Experiment Restart

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

- Simply restart the experiment by simenv.rst without changing any of the SimEnv files describing the experiment and/or the model. The only exception may be the values for the sub-keywords of the keyword ‘experiment’ in the general model configuration file <model>.cfg.

- simenv.rst has the same usage as simenv.run

For parallel models itself see Section 5.9.
• Restart can be launched on an other machine / in an other job class than that of the interrupted experiment.

• Dependent on the experiment log-file <model>.elog, written by the interrupted / previous new-start experiment a single model run from the complete run ensemble in the restart experiment will be
  • Performed if this run has neither a start nor a finish information in the elog-file
  • Not performed if this run has a start and a finish information in the elog-file
  • Performed anew if the run has a start information but no finish information in the elog-file.

• For the latter case a model restart shell script <model>.rst can be provided by the user optionally to prepare restart of this single model run (e.g., by deleting non-SimEnv temporary or output files).

Make sure that in <model>.rst
  • #!/bin/sh is the first line
  • . $SE_HOME/simenv_ini_sh is performed always and as the first SimEnv dot script
  • . $SE_HOME/simenv_end_sh is performed always and as the last SimEnv dot script (see Tab. 5.7 on page 32 and Example 7.4 below).

Make sure that <model>.rst has execute permission by the Unix / Linux command
chmod u+x <model>.rst.

After running $SE_HOME/simenv_get_run_sh the shell script variables run_int and run_char are available in <model>.rst (see Tab. 10.10).

Terminal output from <model>.rst is re-directed to the log-file <model>.nlog.

• Experiment restart works without standard SimEnv experiment preparation. Instead, experiment preparation files and other information from the interrupted experiment will be used.

• For a restart, the optional experiment preparation shell script <model>.ini will be performed only on demand. This request is specified in the configuration file <model>.cfg with the sub-keyword ‘restart_ini’ and its value “yes”. For Python 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” (check Sections 5.5.1 and 5.7.1)

• <model>.cfg will be checked anew for experiment restart. Do not change for a restart any of the information related to the keyword ‘model’ in <model>.cfg.

• Minutes of the restarted experiment will be appended to the log-files <model>.mlog, <model>.nlog, and <model>.elog, respectively from the interrupted experiment.

• Restart can be applied to an experiment several times successively.

• Experiment restart can be performed also as an partial experiment, independently on the partial status of the original model

• Experiment re-start is not possible for the experiment type optimization.

For the model world_sh (check Example 15.6 on page 151) the following contents could be defined for the restart shell script world_sh.rst:

```bash
#!/bin/sh

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

# get run number
. $SE_HOME/simenv_get_run_sh

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

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

Example file: world_sh.rst

Example 7.4 Shell script <model>.rst to prepare model performance during experiment restart
7.5 Experiment Partial Performance

- SimEnv enables to perform an experiment partially by performing only a run slice out of the whole run ensemble.
- Therefore assign appropriate run numbers to the corresponding sub-keywords ‘begin_run’ and ‘end_run’ in <model>.cfg.
- Make sure that begin run number and end run number represent run number from the experiment (including run number 0) and that begin run number ≤ end run number.
- A partial experiment performance is also possible for an experiment restart.
- For more information check Fig. 7.1.
- Experiment partial performance is not possible for the experiment type optimization.
### 7.6 Experiment Related User Shell Scripts and Files

<table>
<thead>
<tr>
<th>Shell script / file</th>
<th>Explanation</th>
<th>Used for</th>
<th>Exist status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;model&gt;.run</strong></td>
<td>model shell script to wrap the model executable&lt;br&gt;Model interface dot scripts at shell script level simenv_<em><em>sh can / have to be applied in &lt;model&gt;.run:&lt;br&gt;• $SE_HOME/simenv_ini_sh has to be performed always and as the first SimEnv dot script simenv</em></em><em>sh&lt;br&gt;• $SE_HOME/simenv_end_sh has to be performed always and as the last SimEnv dot script simenv</em>*_sh&lt;br&gt;• Pre-defined contents for GAMS models</td>
<td>(*)</td>
<td>S R mandatory</td>
</tr>
<tr>
<td><strong>&lt;model&gt;.rst</strong></td>
<td>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&lt;br&gt;• $SE_HOME/simenv_ini_sh has to be performed always and as the first SimEnv dot script simenv_<em><em>sh&lt;br&gt;• $SE_HOME/simenv_end_sh has to be performed always and as the last SimEnv dot script simenv</em></em>_sh&lt;br&gt;• $SE_HOME/simenv_get_run_sh can be used</td>
<td>(**)</td>
<td>R optional</td>
</tr>
<tr>
<td><strong>&lt;model&gt;.ini</strong></td>
<td>model shell script to prepare simulation experiment additionally to standard SimEnv preparation&lt;br&gt;• Experiment will be not performed if return code from this shell script is unequal 0&lt;br&gt;• For experiment re-start &lt;model&gt;.ini will be performed only on request&lt;br&gt;• Pre-defined contents for Python and Gams models</td>
<td>(R)</td>
<td>optional, for Python and GAMS models</td>
</tr>
<tr>
<td><strong>&lt;model&gt;.end</strong></td>
<td>model shell script to clean up simulation experiment from non-SimEnv files&lt;br&gt;• Pre-defined contents for GAMS models</td>
<td>S R</td>
<td>optional</td>
</tr>
</tbody>
</table>

**Files**

<table>
<thead>
<tr>
<th>Shell script / file</th>
<th>Explanation</th>
<th>Used for</th>
<th>Exist status</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>&lt;model&gt;_&lt;run_char&gt;.err</strong></td>
<td>touch such a file during performing the model, in &lt;model&gt;.run and/or in &lt;model&gt;.rst as an indicator to stop the complete experiment after single run &lt;run_char&gt; has been finished</td>
<td>A</td>
<td>optional</td>
</tr>
<tr>
<td>**&lt;model&gt;.jcf_[dis</td>
<td>par</td>
<td>seq]**</td>
<td>model-specific job control file to submit an experiment in distributed, parallel and/or sequential mode by the load leveler LoadL&lt;br&gt;• Copy from general file $SE_HOME/simenv.jcf_[par</td>
</tr>
<tr>
<td><strong>&lt;model&gt;.opt_opt</strong></td>
<td>model-specific control and option file for experiment type Optimization&lt;br&gt;• Copy from general file $SE_HOME/simenv.opt_opt on demand</td>
<td>O</td>
<td>optional</td>
</tr>
</tbody>
</table>

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

(*): shell script applied for<br>R: Restart of an experiment by simenv.rst <model><br>S: Start of an experiment by simenv.run <model><br>file applied for<br>A: All experiment perform. at the login machine or by load leveler submission<br>L: Load leveler experiment submission<br>O: Optimization experiment performance

(**): make sure by the Unix / Linux command chmod u+x <model>.<ext> that the shell script <model>.<ext> has execute permission
First and last single run always refer to the corresponding settings in `<model>.cfg`.

**Fig. 7.1** Flowcharts for performing `simenv.run` and `simenv.rst`

`simenv.run <model>`

1. **start**
2. `<model>.ini`
3. return code = 0 from `<model>.ini`
   - yes
   - no
4. `<run> = first single run`
   - yes
   - no
5. `<run> = last single run`
   - yes
   - no
6. `<model>.run`
7. `<run> = <run> + 1`
8. exists `<model>.<run_char>.err`
   - yes
   - no
9. `<run> = last single run`
   - yes
   - no
10. `<model>.end`
11. **stop**

**simenv.rst <model>**

1. **start**
2. `restart_ini = yes` in `<model>.cfg`
   - yes
   - no
3. `<run> = first single run`
   - yes
   - no
4. single run completed in previous experiments
   - yes
   - no
5. single run unfinished in previous experiments
   - yes
   - no
6. `<model>.rst`
7. `<model>.run`
8. exists `<model>.<run_char>.err`
   - yes
   - no
9. `<run> = last single run`
   - yes
   - no
10. `<model>.end`
11. **stop**
7.7 Saving Experiments

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

- `<model>.out[ all | <run_char> ][ nc | ieee ]` from the model output directory
- `<model>.cfg` from the current workspace
- `<model>.mdf` from the current workspace
- `<model>.edf` from the current workspace
- `<model>.edf_adj` (for optimization) from the current workspace
- `<model>.edf_cf` (for optimization) from the current workspace
- `<model>.elog` (optional) from the current workspace
- `<model>.mlog` (optional) from the current workspace
- `<model>.nlog` (optional) from the current workspace
- `<model>.jcf_[ dis | par | seq ]` (optional) from the current workspace
- `<model>.olog` (optional, for optimization) from the current workspace
- `<model>.opt_opt` (optional, for optimization) from the current workspace

Do not modify after the experiment in

- `<model>.cfg` the information assigned to the keyword ‘model’
- `<model>[ mdf | edf ]` all information including the sequence of the model output variables and/or experiment targets
8 Experiment Post-Processing

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

8.1 General Approach

8.1.1 Post-Processor Results

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

\[ \text{result} = \{ \{ \text{result description} \} \{ \text{[result unit]} \} := \} \text{result expression} \]

\text{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 \text{<ret>} or a sequence of white spaces instead of a result.
For case sensitivity of \text{result} check Tab. 10.11 on page 117.
\text{result description} must not contain an apostrophe character ‘’.
\text{result_unit} characters ‘[‘ and ‘]’ belong to the syntax and are not part of the this document convention as defined in Tab. 1.1
Result description and/or unit together with the separator ‘:=’ have to be specified in the first input line. The result expression itself may follow at the following input line.
\text{result expression} is a chain of SimEnv operators applied to model output variables and/or reference data.
Can be continued on a new input line (continue expression:) if the current input line ends on one of the operators ‘+’, ‘-’, ‘*’, ‘/’, or ‘**’ or on the operand separator ‘,’ in operators.
White spaces are filtered out from the result expression string, also from character arguments.
\text{result description} or \text{result unit} are used to describe the result in the corresponding result output file (see Chapter 12). For the case one of these entities is not specified SimEnv analyses the result expression: For a result expression formed without any operator or only from one operator and using exactly one model output variable and/or one experiment target \text{result description} and/or \text{result unit} is copied from the corresponding information for the sub-keyword ‘descr’ in <model>.mdf (for a model output variable as an operand of this operator) and/or from <model>.edf (for an experiment target as an operand of this operator). The only operator used in this expression must not transform the contents of the operand in general (must be invariant with respect to description and unit). For all other cases \text{result description} is set to the string res_<xy> and \text{result unit} is undefined.
Having a model output variable definition as in Example 5.1 on page 27 then in experiment post-processing:

\[
\text{abs(atmo)} + 3
\]

applies operator abs to atmo and adds 3
(multi-operator result expression)
\[
\text{<result\_description>} = \text{‘res\_<xy>’}
\]
\[
\text{<result\_unit>} \text{ undefined}
\]

Energy [MWh] := \text{abs(atmo)} + 3

as above, but:
\[
\text{<result\_description>} = \text{‘Energy’}
\]
\[
\text{<result\_unit>} = \text{‘MWh’}
\]

Energy [MWh] :=
\[
\text{abs(atmo)} + 3
\]
\[
\text{[MWh]} := \text{abs(atmo)} + 3
\]

as above, but:
\[
\text{<result\_description>} = \text{‘res\_<xy>’}
\]
\[
\text{<result\_unit>} = \text{‘MWh’}
\]

\[
\text{sign(atmo)}
\]

applies operator sign to atmo
(operator sign is not invariant w.r.t. the contents of its operand)
\[
\text{<result\_description>} = \text{‘res<xy>’}
\]
\[
\text{<result\_unit>} \text{ undefined}
\]

\[
\text{abs(atmo)}
\]

applies operator abs to atmo
(operator abs is invariant w.r.t. the contents of its operand)
\[
\text{<result\_description>} = \text{‘aggregated atmospheric state’}
\]
(according to <model>.mdf)
\[
\text{<result\_unit>} = \text{‘without’}
\]
(according to <model>.mdf)

\[
\text{Energy := abs(atmo)}
\]

applies operator abs to atmo
\[
\text{<result\_description>} = \text{‘Energy’}
\]
(according to <model>.mdf)
\[
\text{<result\_unit>} = \text{‘without’}
\]
(according to <model>.mdf)

---

**Example 8.1**  
**Addressing results in experiment post-processing**

### 8.1.2 Operands

Operands in result expressions can be:

- **Model output variables** as defined in <model>.mdf
  In the following abbreviated by **arg**
  Example: `atmo`

- **Experiment targets** as defined in <model>.edf
  In the following abbreviated by **arg**
  Example: `p1`

- **Constants** `<int_val>` or `<real_val>`
  In the following abbreviated by **int_arg** and/or **real_arg**
  Example: `12` and `-12` and `12.34` and `-1.234e+1`

- **Character strings** `<string>`, enclosed in single quotation marks
  In the following abbreviated by **char_arg**
  Example: `'tie_avg'`

- **Operator results**
  In the following abbreviated by **arg**
  Example: `abs(atmo)` and `atmo+3`.

- **Macros** as defined in <model>.mac (see Section 8.7)
  Example: `equ_100yrs m`

- **Wildcard operands** (see Section 8.8)
Example: $&v&$

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

- Dimensionality: $\text{dim(operand)}$
- Extents: $\text{ext(operand,i)}$ with $i=1,\ldots,\text{dim(operand)}$
- Coordinates: $\text{coord(operand,i)}$ with $i=1,\ldots,\text{dim(operand)}$

are assigned to. The dimensionality is the number of dimensions, an extent is related to each dimension and represents the number of elements in that dimension. Extents are always greater than 1. To each dimension a coordinate is assigned to. Coordinates have a name and from all coordinate values the coordinate is defined for a subset is assigned to the extent of the dimension of the operand. Coordinate specification for operands follows that for model output variables. For more information see Section 5.1.

- Operators transform dimensionality, dimensions, and coordinates of the their non-character operator arguments into unique dimensionality, dimensions and coordinates of the operator result (see 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 targets and constants always have a dimensionality of 0.

- Operands of dimensionality 0 and character string operands do not have a coordinate assignment.

### 8.1.3 Model Output Variables

- A variable of dimensionality $n$ corresponds with a $n$-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` (see Tab. 5.3 on page 25) by their
  - Name
  - Dimensionality
  - Extents
  - Coordinate assignment to each dimension
  - Data type (see Tab. 5.4 on page 26).

- Use the service `simenv.chk` to check variables description in model output description file `<model>.mdf`.

- Addressing of model output data fields or parts of it is done in experiment post-processing by corresponding model output variables names.

- For variables with a dimensionality greater than 0 it is possible to address only a part of the whole variable field by
  - Specifying for a dimension an index range $i$ by
    $i = <\text{index\_value}_1, \ldots, \text{index\_value}_n>$

  - Specifying for a dimension a coordinate range $c$ by
    $c = <\text{coordinate\_value}_1, \ldots, \text{coordinate\_value}_n>$

- Index and coordinate ranges are separated from each other by a comma, the sequence of ranges for all dimensions is enclosed in brackets and is appended after the variable name.

- For one variable $c$ and $i$ can be used in mixed mode for different dimensions.
  - $*\text{ denotes the complete range of a dimension.}$
  - $c = *$ is identical to $i = *$ is identical to $*$

- Index addressing
  - $i= \text{ at index addressing}$

- Coordinate addressing
  - $c= \text{ at coordinate addressing}$
In the general SimEnv configuration file <model>.cfg (see Section 10.1 on page 107) 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 27 then in experiment post-processing result expressions can be

\[
\text{atmo} \quad \text{and} \\
\text{atmo}(*,*,*,*) \quad \text{and} \\
\text{atmo}(c=*,*,i=*,*) \quad \text{and} \\
\text{atmo}(c=88:-88,c=-178:178,c=1:16,c=1:20) \quad \text{and} \\
\text{atmo}(i=1:45,i=1:90,i=1:4,i=1:20) \quad \text{and} \\
\text{atmo}(i=1:45,c=-178:178,*,*) \quad \text{and} \\
\text{atmo}(1:45,1:90,1:4,1:20) \quad \text{and (with address_default = index in model.cfg)} \\
\text{atmo}(1:45,c=-178:178,1:4,1:20) \quad \text{and (with address_default = index in model.cfg)} \\
\text{atmo}(*,*,*,c=11:20) \quad \text{addresses all values of last 10 decades} \\
\text{Dimensionality} = 4 \\
\text{Coordinates} = \text{lat, lon, level, time} \\
\text{Extents} = 45, 90, 4, 20 \\
\text{atmo}(*,*,c=11:20) \quad \text{addresses all values of last 10 decades} \\
\text{Dimensionality} = 4 \\
\text{Coordinates} = \text{lat, lon, level, time} \\
\text{Extents} = 45, 90, 4, 10 \\
\text{atmo}(*,*,c=1,c=1) \quad \text{addresses all values of the first decade for level 1} \\
\text{Dimensionality} = 2 \\
\text{Coordinates} = \text{lat, lon} \\
\text{Extents} = 45, 90 \\
\text{atmo}(c=0,*,1,i=20) \quad \text{addresses all values of level 1 for the last decade at equator} \\
\text{Dimensionality} = 1 \\
\text{Coordinates} = \text{lon} \\
\text{Extents} = 90 \\
\text{atmo}(i=23,*,1,i=20) \quad \text{addresses all values of level 1 for the last decade at equator} \\
\text{Dimensionality} = 1 \\
\text{Coordinates} = \text{lon} \\
\text{Extents} = 90 \\
\text{atmo}(c=0,c=2,c=1,c=20) \quad \text{addresses the value for the last decade at} \\
\text{(lat,lon,level,time)} = (0°,2°,1,20) \\
\text{Dimensionality} = 0 \\
\text{Coordinates} = \text{(without)} \\
\text{Extents} = \text{(without)} \\
\text{atmo}(c=0,c=1:9,c=1,c=20) \quad \text{addresses the values for the last decade at} \\
\text{(lat,lon,level,time)} = (0°,2°,1,20) \text{ and (0°,6°,1,20)} \\
\text{Dimensionality} = 1 \\
\text{Coordinates} = \text{lon} \\
\text{Extents} = 2 \\
\text{atmo}(c=0,c=1,c=1,c=20) \quad \text{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 (check Section 8.1.2).

  There are

  - Single-argument operators that replicate dimensionality, dimensions and coordinates from the only argument to the operator result
    Example: \texttt{sin(atmo)}
  - Multi-argument operators that demand a certain relation between dimensionalities, dimensions and coordinates of their arguments
    Example: \texttt{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 (check Tab. 10.9) or form new coordinates from resulting target adjustments
    Example: \texttt{ens(atmo)}
  - SimEnv experiment post-processing operators may have two special types of arguments:
    - Character arguments char\_arg:
      Only character strings enclosed in ‘ ’ are valid as arguments. Some built-in operators (e.g., count) have a pre-defined set of valid character argument strings (e.g., for operator count string numbers all, def, and undef). Some built-in operators allow an empty string (e.g., behav)
      Example: \texttt{count(‘undef’, atmo)}
      \texttt{behav(‘ ‘, atmo)}
    - Integer or real (float) constant arguments int\_arg or float\_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: \texttt{move\_avg(‘0001’, ‘lin’, 3, atmo)}
      \texttt{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: \texttt{hgr\_l(‘1000’, ‘bin\_mid’, 20, 0., 0., atmo)}

  - Operators are generic with respect to the data types of their operands: Each non-character and non-constant argument can be used with operands of all defined data types (see Section 5.1). Internally, arguments of any type are converted to a float representation. This may lead to undefined arguments of type double in float representation.

  - Results of SimEnv experiment post-processing operators are always of the type float.

  - SimEnv post-processing follows the standard approach for description of operators for basic as well as advanced built-in or user-defined operators. 

  Advanced built-in or user-defined operators

  - Have a unique name and a number of operands
  - The sequence of operands is enclosed in parentheses directly after the operator name
  - Operands are separated from each other by a comma.
  - Recursions of the same operator (also for user-defined operators) are possible.
    Example: \texttt{log10(min\_n(3 , min\_n(log10(atmo(*,*,1,c=20)) , 400) , 10*bios\_g))}

  - Elemental operators use the common form of notation:
    Example: \texttt{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.

<table>
<thead>
<tr>
<th>Argument restriction(s) / result description</th>
<th>Argument restriction(s)</th>
<th>Result description (check Section 8.1.2 for syntax)</th>
</tr>
</thead>
</table>
| (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:  
\[
\begin{align*}
\text{dim}(\text{res}) &= \text{dim}(\text{arg}) \\
\text{ext}(\text{res},j) &= \text{ext}(\text{arg},j) \text{ for all } j \\
\text{coord}(\text{res},j) &= \text{coord}(\text{arg},j) \text{ for all } j
\end{align*}
\] |
| (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:  
\[
\begin{align*}
\text{dim}(\text{res}) &= \text{dim}(\text{arg}) \\
\text{ext}(\text{res},j) &= \text{ext}(\text{arg},j) \text{ for all } j \\
\text{coord}(\text{res},j) &= \text{coord}(\text{arg},j) \text{ for all } j
\end{align*}
\] |
| (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:  
\[
\begin{align*}
\text{dim}(\text{res}) &= \text{dim}(\text{arg}) \\
\text{ext}(\text{res},j) &= \text{ext}(\text{arg},j) \text{ for all } j \\
\text{coord}(\text{res},j) &= \text{coord}(\text{arg},j) \text{ for all } j \\
\text{the 0-dimensional argument is applied to each element of the non-zero dimensional argument}
\end{align*}
\] |
| (3) dimensionality, extents and coordinates of the only non-character / non-constant argument can be arbitrary | dim(res) = 0 | |
| (4.1) all non-character / non-constant arguments with same dimensionality, extents and coordinates (*) | dim(res) = 0 | |
| (4) 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 dimensionality, 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:  
\[
\begin{align*}
\text{dim}(\text{res}) &= \text{dim}(\text{arg}) \\
\text{ext}(\text{res},j) &= \text{ext}(\text{arg},j) \text{ for all } j \\
\text{coord}(\text{res},j) &= \text{coord}(\text{arg},j) \text{ for all } j
\end{align*}
\] |
| (6) without arguments | dim(res) = 0 | |

**Tab. 8.1** Classified argument restriction(s) / result description  
(*): for the different levels of checking a coordinate description see below
The requirement for a lot of operators to have same coordinates for same dimensions may restrict application of experiment post-processing especially for hypothesis checking heavily. To enable a broader flexibility with respect to this situation a general solution is provided by SimEnv post-processing: With the sub-keyword 'coord_check' in the general configuration file <model>.cfg three different modes 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.

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

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

**Example 8.3** Checking rules for coordinates
8.2 Built-In Generic Standard Aggregation / Moment Operators

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

<table>
<thead>
<tr>
<th>Generic aggregation and moment operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>maximum of values</td>
</tr>
<tr>
<td>min</td>
<td>minimum of values</td>
</tr>
<tr>
<td>sum</td>
<td>sum of values</td>
</tr>
<tr>
<td>avg</td>
<td>arithmetic mean of values</td>
</tr>
<tr>
<td>var</td>
<td>variance of values</td>
</tr>
<tr>
<td>avgg</td>
<td>geometric mean of values</td>
</tr>
<tr>
<td>avgh</td>
<td>harmonic mean of values</td>
</tr>
<tr>
<td>avgw</td>
<td>weighted mean of values</td>
</tr>
<tr>
<td>hgr</td>
<td>histogram of values</td>
</tr>
<tr>
<td>count</td>
<td>number of values</td>
</tr>
<tr>
<td>maxprop</td>
<td>maximal, suffix related property of values</td>
</tr>
<tr>
<td>minprop</td>
<td>minimal, suffix related property of values</td>
</tr>
</tbody>
</table>

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

For more information check Sections 8.3.3 and 8.4.1.

8.3 Built-In Elemental, Basic, and Advanced Operators

8.3.1 Elemental Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Argument restriction(s) / result description (see Tab. 8.1)</th>
<th>Argument value restriction</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>left parenthesis</td>
<td>-</td>
<td>-</td>
<td>first</td>
</tr>
<tr>
<td>)</td>
<td>right parenthesis</td>
<td>-</td>
<td>-</td>
<td>first</td>
</tr>
<tr>
<td>arg1 ** arg2</td>
<td>exponentiation</td>
<td>(2)</td>
<td>arg1 &gt; 0</td>
<td>second</td>
</tr>
<tr>
<td>arg1 * arg2</td>
<td>multiplication</td>
<td>(2)</td>
<td>arg2 ≠ 0</td>
<td>third</td>
</tr>
<tr>
<td>arg1 / arg2</td>
<td>division</td>
<td>(2)</td>
<td>arg2 ≠ 0</td>
<td>third</td>
</tr>
<tr>
<td>arg1 + arg2</td>
<td>addition (dyadic +)</td>
<td>(2)</td>
<td>arg1 &gt; 0</td>
<td>third</td>
</tr>
<tr>
<td>arg1 – arg2</td>
<td>subtraction (dyadic -)</td>
<td>(2)</td>
<td>arg1 &gt; 0</td>
<td>third</td>
</tr>
<tr>
<td>+ arg</td>
<td>identity (monadic +)</td>
<td>(1)</td>
<td>arg1 &gt; 0</td>
<td>fourth</td>
</tr>
<tr>
<td>– arg</td>
<td>negation (monadic -)</td>
<td>(1)</td>
<td>arg1 &gt; 0</td>
<td>fourth</td>
</tr>
</tbody>
</table>

Tab. 8.3  
**Built-in elemental operators**

- 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Argument restriction(s) / result description (see Tab. 8.1)</th>
<th>Argument value restriction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(arg)</td>
<td>absolute value</td>
<td>(1)</td>
<td></td>
<td>abs(-3) = 3.</td>
</tr>
<tr>
<td>dim(arg1, arg2)</td>
<td>positive difference</td>
<td>(2)</td>
<td>dim(10, 5) = 5.</td>
<td>dim(5, 10) = 0.</td>
</tr>
<tr>
<td>exp(arg)</td>
<td>exponential function</td>
<td>(1)</td>
<td>exp(1.) = 2.7183</td>
<td></td>
</tr>
<tr>
<td>int(arg)</td>
<td>truncation value</td>
<td>(1)</td>
<td>int(7.6) = 7.</td>
<td>int(-7.6) = -7</td>
</tr>
<tr>
<td>log(arg)</td>
<td>natural logarithm</td>
<td>(1)</td>
<td>arg &gt; 0</td>
<td>log(2.7183) = 1.</td>
</tr>
<tr>
<td>log10(arg)</td>
<td>decade logarithm</td>
<td>(1)</td>
<td>arg &gt; 0</td>
<td>log10(10) = 1.</td>
</tr>
<tr>
<td>mod(arg1, arg2)</td>
<td>remainder</td>
<td>(2)</td>
<td>arg2 ≠ 0</td>
<td>mod(10, 4) = 2.</td>
</tr>
<tr>
<td>nint(arg)</td>
<td>round value</td>
<td>(1)</td>
<td>nint(7.6) = 8.</td>
<td></td>
</tr>
<tr>
<td>sign(arg)</td>
<td>sign of value</td>
<td>(1)</td>
<td></td>
<td>sign(-3) = -1.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>sign(0) = 0.</td>
</tr>
<tr>
<td>sqrt(arg)</td>
<td>square root</td>
<td>(1)</td>
<td>arg ≥ 0</td>
<td>sqrt(4) = 2.</td>
</tr>
</tbody>
</table>

### Basic operators

### Trigonometric operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Argument restriction(s) / result description (see Tab. 8.1)</th>
<th>Argument value restriction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(arg)</td>
<td>sine</td>
<td>(1)</td>
<td></td>
<td>sin(0) = 0.</td>
</tr>
<tr>
<td>cos(arg)</td>
<td>cosine</td>
<td>(1)</td>
<td></td>
<td>cos(0) = 1.</td>
</tr>
<tr>
<td>tan(arg)</td>
<td>tangent</td>
<td>(1)</td>
<td>arg ≠ π/2 ± nπ</td>
<td>tan(0) = 0.</td>
</tr>
<tr>
<td>cot(arg)</td>
<td>cotangent</td>
<td>(1)</td>
<td>arg ≠ ± nππ</td>
<td>cot(1,5708) = 0.</td>
</tr>
<tr>
<td>asin(arg)</td>
<td>arc sine</td>
<td>(1)</td>
<td>abs(arg) ≤ 1</td>
<td>asin(0) = 0.</td>
</tr>
<tr>
<td>acos(arg)</td>
<td>arc cosine</td>
<td>(1)</td>
<td>abs(arg) ≤ 1</td>
<td>acos(1) = 0.</td>
</tr>
<tr>
<td>atan(arg)</td>
<td>arc tangent</td>
<td>(1)</td>
<td></td>
<td>atan(0) = 0.</td>
</tr>
<tr>
<td>acot(arg)</td>
<td>arc cotangent</td>
<td>(1)</td>
<td></td>
<td>acot(0) = 1.5708</td>
</tr>
<tr>
<td>sinh(arg)</td>
<td>hyperbolic sine</td>
<td>(1)</td>
<td></td>
<td>sinh(0) = 0.</td>
</tr>
<tr>
<td>cosh(arg)</td>
<td>hyperbolic cosine</td>
<td>(1)</td>
<td></td>
<td>cosh(0) = 1.</td>
</tr>
<tr>
<td>tanh(arg)</td>
<td>hyperbolic tangent</td>
<td>(1)</td>
<td></td>
<td>tanh(0) = 0.</td>
</tr>
<tr>
<td>coth(arg)</td>
<td>hyperbolic cotangent</td>
<td>(1)</td>
<td>arg ≠ 0</td>
<td>coth(3.1416) = 1.</td>
</tr>
</tbody>
</table>

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

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

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

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

- **Appending no suffix:**
  Aggregate the only non-character / non-constant argument
  Result is a scalar (an operator result of dimensionality 0) for all but operators hgr, minprop and maxprop.
  For operator hgr dimensionality of the result is 1, the extent is the specified number of bins for the histogram and the coordinate assigned has the name bin. Coordinate values are equidistant with 1 as the first value and an increment of 1.
  For operators minprop and maxprop dimensionality of the result is 1. For argument dimensionality greater / equal 1 extent of the result is equal to the argument dimensionality. Assigned coordinate name is index. Coordinate values are equidistant with 1 as the first value and an increment of 1. For argument dimensionality 0 result dimensionality is 0.

- **Appending suffix **ₙ** (for **n** arguments)**
  Aggregate an arbitrary number of non-character / non-constant arguments with argument restriction(s) / result description according to (2) in Tab. 8.1 on page 68 element by element
  Currently, only operators minₙ and maxₙ are implemented.
  Result has same dimensionality, extents and coordinates as the arguments

- **Appending suffix **ₙ** (for loop)**
  Aggregate the only non-character / non-constant argument separately for selected dimensions. Dimensions to select are described by an additional loop character argument (corresponds with the group by-clause of the standard query language SQL of relational database management systems).
  Result has a lower dimensionality as the only non-character argument according to the loop character argument.
  For operator hgrₙ, dimensionality is increased additionally by one, the additional extent is the specified number of bins for the histogram and the additional coordinate assigned to has the name bin. Coordinate values are equidistant with 1 as the first value and an increment of 1.

  For operators minpropₙ and maxpropₙ dimensionality is modified in the same manner like for operators minprop and maxprop, respectively.

- **For examples check Section 8.3.5.**
<table>
<thead>
<tr>
<th>Aggregation and moment operator</th>
<th>Argument restriction(s) / result description (see Tab. 8.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>max(arg)</td>
<td></td>
</tr>
<tr>
<td>min(arg)</td>
<td></td>
</tr>
<tr>
<td>sum(arg)</td>
<td></td>
</tr>
<tr>
<td>avg(arg)</td>
<td>(3)</td>
</tr>
<tr>
<td>var(arg)</td>
<td></td>
</tr>
<tr>
<td>avgg(arg)</td>
<td></td>
</tr>
<tr>
<td>avgh(arg)</td>
<td></td>
</tr>
<tr>
<td>avgw(arg1,arg2)</td>
<td>arg2 = weight</td>
</tr>
<tr>
<td>hgr(char_arg1,int_arg2,real_arg3,real_arg4, arg5)</td>
<td>dim(res) = 1 ext(res,dim(res)) = number of bins for char_arg1 = ‘bin_no’ (bin number): coord(res,dim(res)) = name = bin_no values = equidist_end 1(1) number of bins for char_arg1 = ‘bin_mid’ (bin mid): coord(res,dim(res)) = name = bin_mid values = equidist_end 1st bin mid (bin width) number of bins char_arg1 see above int_arg2 = number of bins: 4 ≤ int_arg2 ≤ number_of_values or = 0: automatic determination: number of bins = max(4,number_of_values_of_arg5/10) real_arg3 left bin bound for bin number 1 real_arg4 right bin bound for bin number int_arg2 real_arg3 = real_arg4 = 0.: determine bounds by min(arg5) and max(arg5) min(arg5) = max(arg5): all result values are undefined</td>
</tr>
<tr>
<td>count(char_arg1,arg2)</td>
<td>char_arg1 = [ all</td>
</tr>
<tr>
<td>maxprop(arg)</td>
<td>dim(res) = 1 ext(res,1) = dim(arg) dim(res) = 0 else</td>
</tr>
<tr>
<td>minprop(arg)</td>
<td>return the index of that element of arg where the extreme is reached the first time according to the processing sequence of the argument field arg by the Fortran storage model (see Section 15.7 - Glossary).</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Aggregation and moment operator</th>
<th>Argument restriction(s) / result description (see Tab. 8.1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_n(arg1 ,..., argn)</td>
<td></td>
</tr>
<tr>
<td>min_n(arg1 ,..., argn)</td>
<td>(4)</td>
</tr>
<tr>
<td>maxprop_n(arg1 ,..., argn)</td>
<td>return per result element the argument position (1 ,..., n) where the extreme is reached the first time. Processing sequence starts with arg1.</td>
</tr>
<tr>
<td>minprop_n(arg1 ,..., argn)</td>
<td>(4)</td>
</tr>
</tbody>
</table>

Tab. 8.6  Built-in standard aggregation / moment operators with suffix _n
### Aggregation and moment operator

<table>
<thead>
<tr>
<th>Argument restriction(s) / result description</th>
<th>Argument restriction(s) / result description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim(argi) &gt; 1</td>
<td>dim(arg2) = dim(arg3)</td>
</tr>
<tr>
<td>ext(argi) = arbitrary</td>
<td>ext(arg2,i) = ext(arg3,i)</td>
</tr>
<tr>
<td>dim(res), ext(res,i) according to char_arg1 and argi</td>
<td>arg3 = weight</td>
</tr>
<tr>
<td></td>
<td>dim(res) = 1 + dim(res)</td>
</tr>
<tr>
<td></td>
<td>of all other operators</td>
</tr>
<tr>
<td></td>
<td>ext(res,dim(res)) = number of bins</td>
</tr>
<tr>
<td></td>
<td>for char_arg2 = 'bin_no' (bin number):</td>
</tr>
<tr>
<td></td>
<td>coord(res,dim(res)) = name = bin_no</td>
</tr>
<tr>
<td></td>
<td>values = equidist_end</td>
</tr>
<tr>
<td></td>
<td>1(1) number of bins</td>
</tr>
<tr>
<td></td>
<td>for char_arg2 = 'bin_mid' (bin mid):</td>
</tr>
<tr>
<td></td>
<td>coord(res,dim(res)) = name = bin_mid</td>
</tr>
<tr>
<td></td>
<td>values = equidist_end</td>
</tr>
<tr>
<td></td>
<td>1st bin mid (bin width) number of bins</td>
</tr>
<tr>
<td></td>
<td>char_arg2 see above</td>
</tr>
<tr>
<td></td>
<td>int_arg3 number of bins</td>
</tr>
<tr>
<td></td>
<td>4 ≤ int_arg3 ≤ number_of_values_of_arg6</td>
</tr>
<tr>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td>0: automatic determination</td>
</tr>
<tr>
<td></td>
<td>real_arg4 left bin bound for bin number 1</td>
</tr>
<tr>
<td></td>
<td>real_arg5 right bin bound for bin number</td>
</tr>
<tr>
<td></td>
<td>int_arg3</td>
</tr>
<tr>
<td></td>
<td>real_arg4 = real_arg5 = 0.: determine bounds</td>
</tr>
<tr>
<td></td>
<td>by</td>
</tr>
<tr>
<td></td>
<td>min(arg6) and max(arg6)</td>
</tr>
<tr>
<td></td>
<td>min(arg6) = max(arg6): all result values are undefined</td>
</tr>
<tr>
<td></td>
<td>char_arg22 = [ all</td>
</tr>
<tr>
<td></td>
<td>as above, but: dim(res) is increased by 1 w.r.t. when argi &gt; 0</td>
</tr>
<tr>
<td></td>
<td>ext(res,dim(res)) = dim(arg2)</td>
</tr>
<tr>
<td></td>
<td>coord(res,dim(res)) = name = index</td>
</tr>
<tr>
<td></td>
<td>values = equidist_end 1(1)&quot;n&quot;</td>
</tr>
<tr>
<td></td>
<td>return the indices of those elements of arg2 where the extreme is reached the first time according to char_arg1 and to a Fortran-like processing sequence / storage model (see Section 15.7 - Glossary) of the argument field arg2.</td>
</tr>
</tbody>
</table>

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

The loop character argument char_arg1 is characterised as follows:
- The length of the string is equal to the dimensionality of the non-character argument.
- The string consists of 0 and 1.
- 0 at position n means: aggregate over the corresponding dimension n of the argument.
- 1 at position n means: do not aggregate over the corresponding dimension n of the argument.
- Loop character arguments completely formed of 0 or 1 are forbidden.
### 8.3.4 Advanced Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Argument restriction(s) / result description (see Tab. 8.1)</th>
<th>Argument value restriction</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>classify(int_arg1, real_arg2, real_arg3, arg4)</td>
<td>classify arg4 into int_arg1 classes; potentially restrict classification to interval (real_arg2, real_arg3).</td>
<td>dim(arg4) &gt; 0&lt;br&gt;int_arg1 = number of classes&lt;br&gt;2 ≤ int_arg1 ≤ number of values of arg4&lt;br&gt;0: automatic determination&lt;br&gt;number of classes = ( \max(2, \text{number of values}/10) )&lt;br&gt;real_arg2 = minimum bound for values in class # 1&lt;br&gt;real_arg3 = maximum bound for values in class # int_arg1&lt;br&gt;arg2 = 0. and arg3 = 0.: automatic bound determination</td>
<td></td>
<td>classify((10,0.0,0.,atmo))</td>
</tr>
<tr>
<td>clip(char_arg1, arg2)</td>
<td>clip arg2 according to char_arg1</td>
<td>dim(arg2) &gt; 0&lt;br&gt;dim(res), ext(res,i) depend on char_arg1 and arg2&lt;br&gt;char_arg1 = clip range</td>
<td>dim(arg2) &gt; 0&lt;br&gt;char_arg1 = cumulation indicator per dimension</td>
<td>clip('0','0',1,10', atmo)</td>
</tr>
<tr>
<td>cumul(char_arg1, arg2)</td>
<td>cumulate arg2 according to char_arg1</td>
<td>dim(arg2) &gt; 0&lt;br&gt;char_arg1 = cumulation indicator per dimension</td>
<td></td>
<td>cumul('0001', atmo)</td>
</tr>
<tr>
<td>flip(char_arg1, arg2)</td>
<td>flip arg2 according to char_arg1</td>
<td>dim(arg2) &gt; 0&lt;br&gt;char_arg1 = flip indicator per dimension</td>
<td></td>
<td>flip('0001', atmo)</td>
</tr>
<tr>
<td>get_data(char_arg1, char_arg2, char_arg3, arg4)</td>
<td>get data from an external file</td>
<td>dimensionality, extents and coordinates according to char_arg3 and char_arg4&lt;br&gt;char_arg1 = data file format&lt;br&gt;char_arg2 = data file name&lt;br&gt;char_arg3 = coordinate specification / transformation file name&lt;br&gt;char_arg4 = variable to get from the data file</td>
<td></td>
<td>get_data(‘ascii’, ‘data.asc’, ‘data.def’, variable)</td>
</tr>
<tr>
<td>get_experiment(char_arg1, char_arg2, char_arg3, arg4)</td>
<td>include an other experiment</td>
<td>dim(arg2) &gt; 0&lt;br&gt;char_arg1 = experiment directory&lt;br&gt;char_arg2 = model experimented with&lt;br&gt;char_arg3 = file how to transform result coordinates&lt;br&gt;arg4 = result from the other experiment</td>
<td></td>
<td>get_experiment(‘mod_res’, ‘mod’, ‘mod.trf’, avg(atmo)-400.)</td>
</tr>
<tr>
<td>get_table_fct(char_arg1, arg2)</td>
<td>apply table function with linear interpolation of table char_arg1 to arg2</td>
<td>char_arg1 = file name</td>
<td></td>
<td>get_table_fct(‘table.usr’, atmo)</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Argument restriction(s) / result description (see Tab. 8.1)</td>
<td>Argument value restriction</td>
<td>Example</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----------------------------------------</td>
<td>----------------------------------------------------------------</td>
<td>---------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>if(char_arg1, arg2,arg3,arg4)</td>
<td>conditional if-construct</td>
<td>char_arg1 = comparison operator arg2 = comparator arg3, arg4 = new assignments</td>
<td>(5)</td>
<td>if('&lt;', atmo, 400, atmo)</td>
</tr>
<tr>
<td>mask(char_arg1, arg2,arg3)</td>
<td>mask values of arg2 (set them undefined) by comparing arg2 and arg3 using operator char_arg1</td>
<td>char_arg1 = comparison operator arg2 = comparator arg3, arg4 = new assignments</td>
<td>(5)</td>
<td>mask('&lt;', atmo, 400)</td>
</tr>
<tr>
<td>matmul(arg1, arg2)</td>
<td>matrix multiplication</td>
<td>dim(arg1) = dim(arg2) = dim(res) = 2 ext(res,i) according to matrix multiplication rules</td>
<td></td>
<td>matmul(atmo(<em>,</em>,1,1), transpose('21', atmo(<em>,</em>,1,1)))</td>
</tr>
<tr>
<td>move_avg(char_arg1, char_arg2, int_arg3,arg4)</td>
<td>moving average of arg4</td>
<td>dim(arg4) &gt; 0 char_arg1 = moving average sequence per dimension char_arg2 = average type int_arg3 = running length for average int_arg3 &gt; 1 int_arg3 = 0: automatic determination: = max(3, ext(arg4,i)/20.</td>
<td>(1)</td>
<td>move_avg('001', 'lin', 0, atmo)</td>
</tr>
<tr>
<td>nr_of_runs</td>
<td>number of single runs in the experiment</td>
<td></td>
<td></td>
<td>nr_of_runs()</td>
</tr>
<tr>
<td>rank(char_arg1, arg2)</td>
<td>assign rank numbers to arg2 according to ranking type argument char_arg1</td>
<td>dim(arg2) &gt; 0 arg1 = ranking type [ tie_plain</td>
<td>tie_min</td>
<td>tie_avg ]</td>
</tr>
<tr>
<td>regrid(char_arg1, arg2)</td>
<td>assign completely or partially new coordinates to arg2</td>
<td>(1), but coordinates according to char_arg1 = file how to transform coordinates of arg2 = result to transform coordinates</td>
<td></td>
<td>regrid('mod.trf', atmo_g-13)</td>
</tr>
<tr>
<td>run(char_arg1, arg2)</td>
<td>values of arg 2 for the selected single run number explicitly or implicitly coded in char_arg1</td>
<td>char_arg1 = run number selection = 0 for default run (all experiment types) = &lt;run_number&gt; (for Monte Carlo anal. and loc. sens. anal.: 0 ≤ char_arg1 ≤ number_of_runs) = &lt;filter argument&gt; (for behavioural anal.: same as filter argument of operator behav., check Section 8.4.2)</td>
<td>(1)</td>
<td>run('0', atmo) run('sel_t(pl(4)) ', atmo)</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Argument restriction(s) / result description (see Tab. 8.1)</td>
<td>Argument value restriction</td>
<td>Example</td>
</tr>
<tr>
<td>-----------------------</td>
<td>----------------------------------------------</td>
<td>-------------------------------------------------------------</td>
<td>-----------------------------</td>
<td>------------------------</td>
</tr>
<tr>
<td>transpose(</td>
<td>transpose arg2 according</td>
<td>dim(arg2) &gt; 1</td>
<td>dim(res) = dim(arg2)</td>
<td>transpose('3142',atmo)</td>
</tr>
<tr>
<td>char_arg1,</td>
<td>to sequence in char_arg1</td>
<td>ext(res,i) = ext(arg2,i) (re-sorted)</td>
<td>char_arg1= transpose sequence</td>
<td></td>
</tr>
<tr>
<td>arg2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>undef( )</td>
<td>undefined value</td>
<td></td>
<td>(6)</td>
<td>undef()</td>
</tr>
</tbody>
</table>

### Tab. 8.8 Built-in advanced operators

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

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

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

- The **operator clip** clips an operand arg2 that has dimensionality > 0. The portion to clip from the operand arg2 is described by the argument char_arg1. The argument char_arg1 uses syntax for model output variable addressing (see Section 8.1.3 on page 65). Note, that for all dimensions of argument arg2 lower bound index is 1. This applies also to model output variables where the lower bound index is unequal 1 in the model output description file. In general, extents differ between the result of the operator clip and the argument arg2. Clip reduces the dimensionality of the result with respect to the argument arg2 to clip if the portion to be clipped is limited to one value for at least one dimension. A character argument char_arg1 = '*,..., '* results for operator clip in the identity of argument arg2.

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

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

- With the **operator get_data** data from external files can be included in post-processing. Character argument char_arg1 specifies the file type. Character argument char_arg2 addresses the data file. Character argument char_arg3 is used to define or transform structure information and coordinates from the data file. Argument arg4 holds the variable that is to be extracted from the data file. For restrictions in the path to the directory of the character arguments char_arg2 and char_arg3 check Tab. 11.3. Currently, only ASCII files are supported (char_arg1 = 'ascii'). For ASCII data files the file syntax rules from Section 11.3 are valid. Since the ASCII data file itself does not come with any structure and coordinate information the character argument char_arg3 specifies this information. It follows the same rules as for any coordinate transformation file in Section 11.2. Keywords 'general', 'assign', and 'coordinate' and the appropriate sub-keywords from Tab. 11.5 can be used to structure the data file and to assign coordi-
Having a model output variable definition as in Example 5.1 on page 27 and assuming

da data file data.asc as
# data file with 6 values
10, 20, 30
40, 50, 60

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

general descr structure 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)

It has Dimensionality =2
Coordinates = new_coord, time
Extents = 2, 3

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

\[
\begin{pmatrix}
10 & 30 & 50 \\
20 & 40 & 60
\end{pmatrix}
\]

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

\[
\begin{pmatrix}
10 & 20 & 30 \\
40 & 50 & 60
\end{pmatrix}
\]

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

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

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

- The **operator if** supplies a general conditional if-construct. It operates for each element of the operand arg2 in the following way:
  ```
  if ( condition(arg1,arg2) ) then
    res=arg3
  else
    res=arg4
  endif
  ```
  with condition(arg1,arg2):
  
<table>
<thead>
<tr>
<th>condition</th>
<th>arg2</th>
<th>char_arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg2 &lt; 0</td>
<td></td>
<td>'&lt;'</td>
</tr>
<tr>
<td>arg2 ≤ 0</td>
<td></td>
<td>'&lt;='</td>
</tr>
<tr>
<td>arg2 &gt; 0</td>
<td></td>
<td>'&gt;'</td>
</tr>
<tr>
<td>arg2 ≥ 0</td>
<td></td>
<td>'&gt;='</td>
</tr>
<tr>
<td>arg2 = 0</td>
<td></td>
<td>'='</td>
</tr>
<tr>
<td>arg2 ≠ 0</td>
<td></td>
<td>'!='</td>
</tr>
<tr>
<td>arg2 def</td>
<td></td>
<td>'def'</td>
</tr>
<tr>
<td>arg2 undef</td>
<td></td>
<td>'undef'</td>
</tr>
</tbody>
</table>

- The **operator mask** supplies a method to mask values. It operates for each element of the operand arg2 in the following way:
  ```
  if ( condition(arg1,arg2,arg3) ) then
    res=undef( )
  else
    res=arg2
  endif
  ```
  with condition(arg1,arg2,arg3):
  
<table>
<thead>
<tr>
<th>condition</th>
<th>arg2</th>
<th>char_arg1</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg2 &lt; arg3</td>
<td></td>
<td>'&lt;'</td>
</tr>
<tr>
<td>arg2 ≤ arg3</td>
<td></td>
<td>'&lt;='</td>
</tr>
<tr>
<td>arg2 &gt; arg3</td>
<td></td>
<td>'&gt;'</td>
</tr>
<tr>
<td>arg2 ≥ arg3</td>
<td></td>
<td>'&gt;='</td>
</tr>
<tr>
<td>arg2 = arg3</td>
<td></td>
<td>'='</td>
</tr>
<tr>
<td>arg2 ≠ arg3</td>
<td></td>
<td>'!='</td>
</tr>
</tbody>
</table>

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

- The **operator move_avg** performs a moving average operation successively for selected dimensions of the argument arg4.
  For a vector (a1, a2, ..., a_len) the moving average of running length rl is a vector (ma1, ma2, ..., ma_len) with elements
\[ m_{ai} = \frac{1}{\sum_{j=\max(1,i-rl+1)}^{\min(1,i-rl+1)}} \sum_{j=\max(1,i-rl+1)}^{\min(1,i-rl+1)} w_{ij} \cdot a_j \]

where \( w_{ij} \) are weights. Value \( m_{ai} \) is averaged from the \( rl \) values \( a_i, a_{i-1}, \ldots, a_{i-rl+1} \). Accordingly, the first \( rl-1 \) values \( m_{a1}, m_{a2}, \ldots, m_{a_{rl-1}} \) are averaged from less than \( rl \) values.

For the linear moving average the weight is \( w_{ij} = 1 \) and \( \sum_{j=\max(1,i-rl+1)}^{\min(1,i-rl+1)} w_{ij} = \min(rl,i) \), for the exponential moving average the weight is \( w_{ij} = e^{-\frac{|i-j|}{rl}} \).

While the moving average is normally applied to time-dependent one-dimensional data vectors the operator \text{move\_avg} \ allows processing of multi-dimensional data fields in a general and succesive manner.

For example, if \( \text{arg4} \) is the three-dimensional variable \text{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 \( (\text{lat1} = 1, \ldots, \text{lat}) \times (\text{lon1} = 1, \ldots, \text{lon}) = \text{lat*lon moving averages will be performed for the vector} \)

\[ ( \text{bios(lat1,lon1,1)}, \text{bios(lat1,lon1,2)} , \ldots, \text{bios(lat1,lon1,time)} ) \].

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

\[ ( \text{tmp(1,lon1,time)}, \text{tmp(2,lon1,time1)}, \ldots, \text{tmp(lat,lon1,time1)} ) \].

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

\text{move\_arg('201', 'lin', 0, bios)}.

The character argument \text{char\_arg1} supplies those dimensions that are to be involved in the moving average operation. If the \( n \)-th digit of \text{char\_arg1} is a digit \( > 0 \) then the moving average for dimension \( n \) of argument \text{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 \text{arg1} is 0 then the moving average for the dimension \( n \) of \text{arg4} will not be performed.

Keep in mind that the sequence of moving averages for single coordinates influences the result of the operator.

- The \text{operator} \text{nr\_of\_runs} \ returns the number of performed single runs of the current post-processed experiment without the run number 0 of the nominal constellation. It does not have an argument.

- The \text{operator} \text{rank} \ transforms all values of the operand \text{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 \text{char\_arg1} determines how to rank ties, i.e., values of \text{arg2} that are identical or have a maximum absolute difference of \( 10^{-6} \).

Assume an argument \text{arg2} with 6 values \( (\text{4.}, \text{2.}, \text{4.}, \text{4.}, \text{8.}) \).

Then \text{char\_arg1} = \text{'tie\_plain'} \ returns ranks \( (\text{2}, \text{1}, \text{2}, \text{2}, \text{3}) \)

four times rank 2; next rank is 3, does not take into account the number of identical values

\text{char\_arg1} = \text{'tie\_min'} \ returns ranks \( (\text{2}, \text{1}, \text{2}, \text{2}, \text{6}) \)

four times rank 2; next rank is 6, takes into account the number of identical values

\text{char\_arg1} = \text{'tie\_avg'} \ returns ranks \( (\text{3.5}, \text{1}, \text{3.5}, \text{3.5}, \text{3.5}, \text{6}) \)

four times mean rank 3.5 = \( (2+3+4+5)/4 \); next rank is 6, takes into account number of identical values

- The \text{operator} \text{regrid} \ can be used to assign new coordinates to argument \text{arg2}. Character argument \text{char\_arg1} is the name of the coordinate transformation file that holds the information how to transform the coordinates. The keyword ‘modify’ and the corresponding sub-keywords are not allowed. For syntax of coordinate transformation files check Section 11.2. For restrictions in the path to the directory of the character arguments \text{char\_arg1} check Tab. 11.3.
The **operator run** selects a single run from the run ensemble. The operator run must not contain experiment specific (multi-run) operators as operands, since these operators may refer to the operator run. Additionally, run must not contain itself as an argument.

The character argument `char_arg1` can hold the run number string explicitly. Explicit run number string in character argument `char_arg1` is allowed for Monte Carlo and local sensitivity analyses. Additionally, for behavioural and local sensitivity analysis a run number unequal 0 can be selected implicitly by applying a filter of the corresponding operators (see Sections 8.4.2 and 8.4.4) as `char_arg1` of the operator run.

The file `<model>.edf_adj` holds the target values to be adjusted to the default values for the current experiment. Run number `n` corresponds with record number `n+1` of this file. Single run number 0 corresponds with the default single run 0. For more information on `<model>.edf_adj` check Section 6.1 on page 43. For examples see Example 8.6 and Example 8.7.

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 with the re-ordered sequence of the operator result extents.

A character argument `char_arg1 = '123...'` results for the operator transpose in the identity of argument `arg2`.

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

For examples of all 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 27 and assuming `address_default=coordinate in world_*`.cfg then in experiment post-processing

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>atmo_g+2*atmo_g</code></td>
<td>value of result <code>3*atmo_g</code>&lt;br&gt;Dimensionality = 1&lt;br&gt;Coordinates = time&lt;br&gt;Extents = 20</td>
</tr>
<tr>
<td><code>sqrt(atmo_g)</code></td>
<td>square root of <code>atmo_g</code>&lt;br&gt;Dimensionality = 1&lt;br&gt;Coordinates = time&lt;br&gt;Extents = 20</td>
</tr>
</tbody>
</table>
| `clip('i=23,*,1,19:20',atmo)` | last two decades for level 1 at equator<br>equivalent with `atmo(i=23,*,[1,19:20])`
Dimenstonality = 2
Coordinates = lon, time
Extents = 90, 2 |
| `atmo - get_experiment('./other_dir','other_model',' ',atmo)` | Difference for `atmo` between the current experiment and another model `other_model`, located in directory `./other_dir` without application of a coordinate transformation file
Dimensionality = 4
Coordinates = lat, lon, level, time
Extents = according to definition of `atmo` in `other_model` |
| `get_table_fct('world.dat_tab',atmo)` | Operator table_fct with table `world.dat_tab` applied to each element of `atmo`
Dimensionality = 4
Coordinates = lat, lon, level, time
Extents = 45, 90, 4, 20 |
### Example 8.5  
**Experiment post-processing with advanced operators**

#### 8.4 Built-In Experiment Specific Operators

- Experiment specific operators are to navigate and process in the experiment space.
- Experiment specific operators must not be applied recursively.
- Addressing a variable within an experiment specific operator normally results in application of the operator on the whole run ensemble or parts of it and in aggregating across the run ensemble according to the operator.

```plaintext
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

eqv(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, 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
```
• Addressing a variable outside an experiment specific operator results in application of the basic, advanced and/or user-defined operator on the variable for the default run number 0 of the experiment.
• If the dimensionality of an operator result is higher than that of one of its operands the additional dimensions of the result are appended to the dimensions of the operand. Examples for such operators are ens (for Monte Carlo analysis post-processing) and behav (for certain constellations of behavioural analysis post-processing).

### 8.4.1 Standard Aggregation / Moment Operators

Tab. 8.9 summarises multi-run standard aggregation / moment operators for behavioural analysis, Monte Carlo analysis and optimization. They work on the whole run ensemble (for Monte Carlo analysis and optimization) or parts of it (for certain constellations of behavioural analysis post-processing). They are used with suffix _e for Monte Carlo analysis and optimization and without any suffix for behavioural analysis. For a definition of these operators check Tab. 8.2 on page 70.

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

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

- A single run can be selected from the run ensemble
- The complete run ensemble can be addressed
- Sub-spaces from the experiment space can be addressed and
- Sub-spaces can be projected by aggregation and moment operators
dependent on the way the experiment target space was to be scanned according to the sub-keyword ‘comb’
in the experiment description file.

To show the power of the operator behav the simple experiment layouts as described in Fig. 4.3 on page 14
are used as examples.

- With behav it is possible to address for any operand a single run out of the run ensemble by fixing values
  of experiment targets p1 and p2 (for Fig. 4.3 (a)), a value of the parallel targets p1 or p2 (for Fig. 4.3 (b)),
  and values of targets p3 and p1 or p2 (for Fig. 4.3 (c)). Dimensionality and extents of the operator result
  is the same as that of the operand.

- Without any selection in the target experiment space (p1,p2) and/or (p1,p2,p3) the dimensionality of the
  operator result is formed from the dimensionality of the operand enlarged by the dimensionality of the
  experiment space. Two additional dimensions are appended to the operand for Fig. 4.3 (a), one addi-
  tional dimension for Fig. 4.3 (b), and two additional dimensions for Fig. 4.3 (c). For the latter two cases it
  is important which of the axis p1 and p2 is used for further processing and/or output of the operator re-
  sult. The extents of the appended dimensions are determined by the number of target adjustments.

- As a third option it is possible to select only a sub-space out of the experiment space to append to the
  operand. For Fig. 4.3 (a) this could be the sub-space formed from the first until the third adjustment
  value of p1 and all adjustment values of p2 between 3 and 7. Dimensionality of the operator result in-
  creases by 2 and extents of these additional dimensions are 3 and 2 with respect to the corresponding
  Example 6.1 (a) in Section 6.2.2 on page 46.

- The operator behav also enables to aggregate operands in the experiment space. In correspondence
  with the example in the last bullet point for Fig. 4.3 (a) the operand could be aggregated (e.g., averaged)
  over the first until the third adjustment 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 ad-
  justment values of p2 between 3 and 7. Consequently, the result has the same dimensionality as the
  operand of behav. Sequence of performing aggregations is important.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Argument restriction(s) / result description</th>
<th>Argument value restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>behav(char_arg1, arg2)</td>
<td>navigation and aggregation in the experiment space for arg2 according to char_arg1</td>
<td>char_arg1 = selection / aggregation filter according to Tab. 8.14 dim(res) = dim(arg2) + appended dimensions according to char_arg1</td>
<td></td>
</tr>
</tbody>
</table>

Tab. 8.10 **Experiment specific operator for behavioural analysis**
The following rules hold for the operator behav:

- Generally, by the filter argument arg1 those runs from the run ensemble are selected and/or aggregated (here interpreted as filtered) that are used for the formation of the result. Consequently, if no filter is specified all runs are used:

  \( \text{behav(' ',atmo.g)} \)

  The select operator has to be specified only if values are to be restricted by a corresponding target value range.

  For the aggregation and the select operator the target value type is redundant if the value range represents the full range of values by \(<\text{target_name}>\) or \(<\text{target_name}>(*)\):

  \( \text{sel}(p1) = \text{sel}(p1(*)) = \text{sel}_i(p1) = \text{sel}_v(p1) = \text{sel}_t(p1) \) and all are redundant.

- The show-operator can be used to force a certain experiment target to be used in the result of the operator behav if this target is used in parallel with other targets. By default, the first target of a parallel target sub-space as declared in the comb-line of the experiment description file is used in the behav-result.

- Aggregation operators reduce dimensionality of the covered experiment target space in the behav-result. The sequence of aggregation operators the first argument of the operator behav influences the result: Computation starts with the first aggregation operator and ends with the last:

  \( \text{avg}(p1), \text{min}(p2) \) normally differs from \( \text{min}(p2), \text{avg}(p1) \)

- An unused experiment target in the selection and aggregation filter contributes with an additional dimension to arg2 to the result of the operator behav. The extent of this additional dimension corresponds with the number of adjustments to this target in the experiment description file.

- A target that is restricted by any of the select operators also contributes with an additional dimension to the result of the operator behav if the number of selected values is greater than 1. The extent of the additional dimension corresponds with the number of selected values of this target by the select operator. Consequently, an empty character string arg1 forces to output the operand arg2 over the whole target space of the experiment.

- The name of the coordinate that is assigned to an additional dimension is the name of the corresponding target. Coordinate description and coordinate unit (see 5.1 on page 21) are associated with the corresponding information for the target from the experiment description file. Coordinate values are formed from resulting target values. For strictly ordered target adjustments in the experiment description file and finally for strictly ordered resulting target values the coordinate values are ordered accordingly in an increasing or decreasing manner. Unordered target adjustments and finally unordered target values are ordered in an increasing manner for coordinate usage.
The result of the operator behav is always arranged according to ascending coordinate values for all additional dimensions.

- Independently from the sequence of the applied aggregation-, select- and show-operators the targets that contribute to additional dimensions of the result of the operator behav are appended to the dimensions of the operand arg2 of behav according to the sequence they are declared in the experiment description file (and not to the sequence they are used in the comb-line of the experiment description file). From parallel changing targets that target is used in this sequence that is addressed explicitly or implicitly by the show-operator.
- For experiment targets that are changed in the experiment in parallel, that increase dimensionality of the result and where a show-operator is missing the first target from this parallel sub-space in the comb-line is used in the result.
- For experiments that use an adjustment file (<model>.edf: specific comb file ...) instead of adjustment definitions (<model.edf>: specific [ default | <combination> ]) all experiment targets are assumed to be adjusted in parallel.

Having a model output variable definition as in Example 5.1 on page 27 and assuming address_default = coordinate in world_.cfg
Assume the experiment layout in Example 6.1 (c) on page 46 and the corresponding experiment description file (c) from Example 6.1 on page 46
then in result-processing

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

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

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

behav('sel_i(p2(1:3)),sel_v(p3(1:2))',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 = 3.3, 4.5  
Dimensionality = 4  
Coordinates = lat , lon , p2 , p3  
Extents = 45 , 90 , 3 , 2

behav('avg_i(p2(1:3)),sel_i(p3(2:3))',atmo(*,*,1,*))  
mean of atmo for level 1 and for runs with p2 =1, 2, 3  
for each value of p3 = 4.5, 7.2  
Dimensionality = 4  
Coordinates = lat , lon , time , p3  
Extents = 45 , 90 , 20 , 2

behav('min(p2),max(p3)',avg(atmo(*,*,1,19:20)))  
determine single minima of avg(atmo) for level 1 and the  
last two decades for each value of p2  
afterwards determine from that the maximum over all p3.  
Dimensionality = 0  
Coordinates = (without)  
Extents = (without)
```
behav('max(p3),min(p2)',avg(atmo(*,*,1,19:20)))

Result differs normally from min(p2),max(p3)
(Previous result expression)

behav('count(def,p3),sel_i(p2=1)',bios(*,*,20))/3

determine single numbers of defined values of
bios for last decade for runs with p2=1.
Result consists of values 0 (for water) and 1 (for land)
Dimensionality = 2
Coordinates = lat , lon
Extents = 36 , 90

behav(' ',atmo(*,*,1,20)-run('sel_i(p1(1)),sel_i(p3(3))',
atmo(*,*,1,20)))

deviation of the last time step of atmo for level 1
from the run with p1=1, p2=1, p3=3.3
dependent on (p2,p1) and p3
Dimensionality = 4
Coordinates = lat , lon , p2 , p3
Extents = 45 , 90 , 4 , 3

Example file: world.post_c

Example 8.6  Experiment post-processing operator behav for behavioural analysis

8.4.3 Monte Carlo Analysis

Tab. 8.12 shows experiment specific operators for Monte Carlo analysis that can be used in post-processing besides the general multi-run aggregation operators listed in Tab. 8.9 on page 83 and supplemented with a suffix _e.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Argument restriction(s) / result description (see Tab. 8.1)</th>
<th>Argument value restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnf(real_arg1, _arg2)</td>
<td>positive distance of confidence measure from mean avg_e(arg2)</td>
<td>real_arg1 error probability</td>
<td>arg1 = [0.001</td>
</tr>
<tr>
<td>cor(arg1, arg2)</td>
<td>correlation coefficient between arg1 and arg2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cov(arg1, arg2)</td>
<td>covariance between arg1 and arg2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ens(arg)</td>
<td>whole Monte Carlo run ensemble</td>
<td>dim(res) = dim(arg)+1 ext(res,dim(res)) = number_of_runs coord(res,dim(res)) = name = run values = equidist_end 1(1) number_of_runs</td>
<td></td>
</tr>
<tr>
<td>krt(arg)</td>
<td>kurtosis (4th moment)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>med(arg)</td>
<td>median</td>
<td></td>
<td></td>
</tr>
<tr>
<td>qnt(real_arg1, _arg2)</td>
<td>quantile of arg2</td>
<td>real_arg1 quantile value</td>
<td>0. ≤ arg1 ≤ 100.</td>
</tr>
<tr>
<td>reg(arg1, arg2)</td>
<td>linear regression coefficient to forecast arg2 from arg1: arg2 = reg(arg1, arg2)*arg1 + n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Argument restriction(s) / result description (see Tab. 8.1)</td>
<td>Argument value restriction</td>
</tr>
<tr>
<td>----------------</td>
<td>---------------------------------------------------</td>
<td>-------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>rng(arg)</td>
<td>range = max_e(arg) - min_e(arg)</td>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td>skw(arg)</td>
<td>skewness (3rd moment)</td>
<td></td>
<td>(1)</td>
</tr>
<tr>
<td>stat_full(</td>
<td>full basic statistical measures of arg5</td>
<td>dim(res) = dim(arg)+1</td>
<td>arg1, arg2 = [ 0.001</td>
</tr>
<tr>
<td>real_arg1,</td>
<td></td>
<td>ext(res,dim(res)) = 10</td>
<td>arg1 &lt; arg2 error probability for confidence distance measure</td>
</tr>
<tr>
<td>real_arg2,</td>
<td></td>
<td>coord(res,dim(res)) =</td>
<td>0. ≤ arg3 &lt; arg4 ≤ 100. quantile values</td>
</tr>
<tr>
<td>real_arg3,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real_arg4,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>real_arg5)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>stat_red(</td>
<td>reduced basic statistical measures of arg3</td>
<td>dim(res) = dim(arg)+1</td>
<td>arg1, arg2 = [ 0.001</td>
</tr>
<tr>
<td>real_arg1,</td>
<td></td>
<td>ext(res,dim(res)) = 7</td>
<td>arg1 &lt; arg2 error probability for confidence distance measure</td>
</tr>
<tr>
<td>real_arg2,</td>
<td></td>
<td>coord(res,dim(res)) =</td>
<td>0. ≤ arg3 &lt; arg4 ≤ 100. quantile values</td>
</tr>
<tr>
<td>real_arg3)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 8.12** 
*Experiment specific operators for Monte Carlo analysis (without standard aggregation / moment operators)*

The following explanations hold for the operators in Tab. 8.12:

- The operators **stat_full** and **stat_red** supply basic statistical measures for their last argument. Both operators are stand-alone operators: They must not be operands of any other operator. Contrary, their last argument can be composed from other non-multi-run operators. To store the statistical measures, dimensionality of both operators is that of their last argument, appended by an additional dimension with an extent of 10 and/or 7. Appended coordinate description is pre-defined by SimEnv (check Tab. 10.9).

These ten data fields (for operator stat_full) and/or seven data fields (operator stat_red) correspond with 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 real_arg1
7. Run ensemble positive distance of confidence measure from run ensemble mean for real_arg2

**Only for operator stat_full:**

8. Run ensemble median
9. Run ensemble quantile of quantile value real_arg3
10. Run ensemble quantile of quantile value real_arg4

The operator stat_red was introduced because computation of the median and quantiles consumes a lot of auxiliary storage space. For the definition of the statistical measures check the corresponding single operators in Tab. 8.9 and Tab. 8.12. Both operators were designed for application of an appropriate visualization technique in result evaluation in future.
Having a model output variable definition as in Example 5.1 on page 27 and assuming address_default=coordinate in world_*.cfg
Assume the Monte Carlo experiment from Example 6.2 (e) on page 49 then in experiment post-processing

\[ \text{avg}_e(p1*atmo(*,*,1,19:20)) \]  
-global run ensemble mean of \( p1 \cdot \text{atmo} \) for level 1 and the last two decades  
Dimensionality = 3  
Coordinates = lat, lon, time  
Extents = 45, 90, 2

\[ \text{avg}(\text{atmo}(*,*,19:20)) \]  
-global mean of \( \text{atmo} \) for level 1 and the last two decades for run number 0  
Dimensionality = 0  
Coordinates = (without)  
Extents = (without)

\[ \text{ens}(\text{atmo}(*,*,1,20)) \]  
-run ensemble values of \( \text{atmo} \) for level 1 and the last decade  
Dimensionality = 3  
Coordinates = lat, lon, run  
Extents = 45, 90, 250

\[ \text{minprop}_e(\text{atmo}(*,*,1,19:20)) \]  
-run ensemble run number for level 1 and the last two decades where the minimum of \( \text{atmo} \) is reached the first time  
Dimensionality = 3  
Coordinates = lat, lon, time  
Extents = 45, 90, 2

\[ \text{var}_e(\text{atmo}(*,*,1,19:20)) - \text{atmo}(*,*,1,19:20) \]  
-anomaly for run ensemble variance from the nominal run for level 1 the last two decades  
Dimensionality = 3  
Coordinates = lat, lon, time  
Extents = 45, 90, 2

\[ \text{var}_e(\text{atmo}(*,*,1,19:20)) - \text{run}('0',\text{atmo}(*,*,1,19:20)) \]  
-global run ensemble variance of the anomaly of \( \text{atmo} \) 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

\[ \text{hgr}_e('bin\_no',0,0.,0.,\text{min}\_l('10',\text{atmo}(20:-20,*,1,20))) \]  
-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 \( \text{hgr}_e \).  
Dimensionality = 2  
Coordinates = lat, bin_no  
Extents = 11, 25

\[ \text{stat\_full}(0.01,0.05,25,75, \text{min}\_l('10',\text{atmo}(20:-20,*,1,20))) \]  
-full basic statistical measures for the zonal tropical minima of \( \text{atmo} \) for level 1 and the last decade  
Dimensionality = 2  
Coordinates = lat, stat_measure  
Extents = 11, 10

Example file: world.post_e

Example 8.7  Experiment post-processing operators for Monte Carlo analysis
8.4.4 Local Sensitivity Analysis

Tab. 8.13 shows the experiment specific operators for local sensitivity analysis that can be used in post-processing. For a definition of these operators check Tab. 4.3 on page 17.

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Argument restriction(s) / result description</th>
<th>Argument value restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>sens_abs(...)</td>
<td>absolute sensitivity measure for arg2 according to char_arg1</td>
<td>arg1 = selection / aggregation filter</td>
<td>dim(res) = dim(arg2) + appended dimensions according to char_arg1</td>
</tr>
<tr>
<td>sens_rel(...)</td>
<td>relative sensitivity measure for arg2 according to char_arg1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lin_abs(...)</td>
<td>absolute linearity measure for arg2 according to char_arg1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lin_rel(...)</td>
<td>relative linearity measure for arg2 according to char_arg1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sym_abs(...)</td>
<td>absolute symmetry measure for arg2 according to char_arg1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sym_rel(...)</td>
<td>relative symmetry measure for arg2 according to char_arg1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tab. 8.13 Experiment specific operators for local sensitivity analysis

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

- Generally, by the filter argument char_arg1 those runs from the run ensemble are selected (here interpreted as filtered) that are used for the formation of the result.

Consequently, if no filter is specified all runs are used:
sens_abs(' ',atmo_g)
The filter operator has to be specified only if values are to be restricted by corresponding target values, increment values and/or sign ranges.

- For the above three select operators selt, seli and sels the value type is redundant if the value range represents the full range of values by [ selt | seli | sels ] (*):
  
selt(*) = selt.n(*) = selt.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 value range is specified by position indices those targets are selected for selt and/or those increments are selected for seli that correspond with the specified position indices. Position indices are assigned from index 1 to the targets and or increments according to their specification sequence in the corresponding experiment description file <model>.edf.
- If more than one target, increment value and/or sign was selected by the filter argument arg1 it contributes with an additional dimension to the result of the local sensitivity operator:
  
  - For targets an additional dimension target_sequ
  - For increments an additional dimension incr
  - For signs an additional dimension sign

is appended to the dimensions of the argument arg2 to form the result of the local sensitivity operator. The extent of this additional dimension corresponds with the defined and/or selected number of targets, increment values and/or signs. For a definition of the additional dimensions check Tab. 10.9. Firstly, dimension target_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 27 and assuming address_default=coordinate in <model>.cfg
Assume the experiment description file (f) from Example 6.3 on page 50
then in result-processing

sens_abs( ' ', atmo_g )
absolute sensitivity measure for atmo_g
for all targets, increments and signs
Dimensionality = 4
Coordinates = time, target_sequ, incr, sign
Extents = 20, 3, 4, 2

sens_rel( 'sels_n(+),selt_i(1)', atmo_g )
relative sensitivity measure for atmo_g
for target p1 and all positive increments
Dimensionality = 2
Coordinates = time, incr
Extents = 20, 4

sens_abs( 'seli_v(0.001:0.05)', atmo_g )
absolute sensitivity measure for atmo_g
for all targets, increment values 1 to 3 and all signs
Dimensionality = 4
Coordinates = time, target_sequ, incr, sign
Extents = 20, 3, 3, 2

lin_abs( 'seli_v(0.001:0.05)', atmo_g )
absolute linearity measure for atmo_g
for all targets and increment values 1 to 3
Dimensionality = 3
Coordinates = time, target_sequ, incr, sign
Extents = 20, 3, 3

Example file: world.post_f

Example 8.8  Experiment post-processing operators for local sensitivity analysis
8.4.5 Optimization

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

While the single run that corresponds with the optimal cost function can be post-processed in the single-run modus, the whole experiment can be post-processed as a Monte Carlo analysis. Keep in mind that the targets do not follow a pre-defined distribution.
8.5 User-Defined and Composed Operators / Operator Interface

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

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

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

8.5.1 Declaration of User-Defined Operator Dynamics

User-defined operators consist of a declarative and a computational part, that are described in one source file in two C/C++ or Fortran functions (see Tab. 8.15):

- 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.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Function description</th>
<th>Inputs / outputs / function value description</th>
<th>Inputs / outputs / function value description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>simenv_check_user_def_operator()</code></td>
<td>check consistency of operator arguments and define dimensionality and dimensions of the result</td>
<td><code>integer*4 simenv_check_user_def_operator(function value)</code></td>
<td>return code = 0 ok ≠ 0 inconsistency between operands</td>
</tr>
<tr>
<td><code>simenv_compute_user_def_operator(res)</code></td>
<td>compute result of the operator in dependency on operands</td>
<td><code>real*4 res(1) (output)</code></td>
<td>result vector of the operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>integer*4 simenv_compute_user_def_operator(function value)</code></td>
<td>return code = 0 ok ≠ 0 user-defined interrupt of calculation</td>
</tr>
</tbody>
</table>

Operator results of a dimensionality > 1 have to be stored to the field `res` using the Fortran storage model (see Section 15.7 - Glossary).

Tab. 8.15 Operator interface functions for the declarative and computational part
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 greater than 5.

Tab. 8.16 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.

<table>
<thead>
<tr>
<th>Function name</th>
<th>Function description</th>
<th>Inputs / outputs / function value</th>
<th>Inputs / outputs / function value</th>
</tr>
</thead>
<tbody>
<tr>
<td>simenv_get_char_arg_[ f</td>
<td>c ] (iarg, char)</td>
<td>get string and string length of a character argument</td>
<td>integer*4 iarg (input)</td>
</tr>
<tr>
<td>simenv_get_char_arg_[ f</td>
<td>c ] (function value)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simenv_get_char_arg_[ f</td>
<td>c ] (iarg, iext)</td>
<td>Iarg &gt; 0: get dimensionality and extents of an argument Iarg = 0: get dimensionality and extents of the result</td>
<td>integer*4 iarg (input)</td>
</tr>
<tr>
<td>simenv_get_dim_arg_[ f</td>
<td>c ] (function value)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>simenv_get_len_arg_[ f</td>
<td>c ] (iarg)</td>
<td>Iarg &gt; 0: get length of an argument Iarg = 0: get length of the result</td>
<td>integer*4 iarg (input)</td>
</tr>
<tr>
<td>simenv_get_nr_arg_[ f</td>
<td>c ] ( )</td>
<td>get number of arguments of the current operator</td>
<td>integer*4 simenv_get_nr_arg_[ f</td>
</tr>
<tr>
<td>Function Name</td>
<td>Function Description</td>
<td>Inputs / Outputs / Function Value</td>
<td>Inputs / Outputs / Function Value</td>
</tr>
<tr>
<td>---------------</td>
<td>----------------------</td>
<td>-----------------------------------</td>
<td>-----------------------------------</td>
</tr>
<tr>
<td>simenv_get_type_arg_[ f</td>
<td>c ] ( iarg )</td>
<td>larg &gt; 0: get data type of an argument iarg = 0: get data type of the result</td>
<td>integer*4 iarg (input)</td>
</tr>
<tr>
<td>simenv_get_co_chk_modus_[ f</td>
<td>c ] ( )</td>
<td>get level of coordinate check for arguments according to &lt;model&gt;.cfg</td>
<td>integer*4 simenv_get_co_chk_modus_[ f</td>
</tr>
<tr>
<td>simenv_get_co_arg_[ f</td>
<td>c ] ( iarg, ico_nr, ico_beg_pos, co_name )</td>
<td>get formal coordinate numbers and formal coordinate begin value positions of an argument</td>
<td>integer*4 iarg (input)</td>
</tr>
<tr>
<td>simenv_get_co_val_[ f</td>
<td>c ] ( ico_nr, ico_pos, co_val )</td>
<td>get for a coordinate a coordinate value at a specified position</td>
<td>integer*4 ico_nr (input)</td>
</tr>
<tr>
<td>simenv_get_co_arg_[ f</td>
<td>c ] ( ico_pos )</td>
<td>Application of this function in simenv_check_user_def_operator for coordinate bin_mid results in an error.</td>
<td>integer*4 ico_pos (input)</td>
</tr>
<tr>
<td>simenv_get_co_val_[ f</td>
<td>c ] ( ico_nr, co_val )</td>
<td></td>
<td>real*4 co_val (output)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 simenv_get_co_val_[ f</td>
<td>c ] (function value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 0 ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 1 ico_pos out of range</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 2 storage exceeded</td>
</tr>
<tr>
<td>Function name</td>
<td>Function description</td>
<td>Inputs / outputs / function value</td>
<td>Inputs / outputs / function value</td>
</tr>
<tr>
<td>---------------</td>
<td>---------------------</td>
<td>----------------------------------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>simenv_chk_2args_[f</td>
<td>c] (iarg1, iarg2)</td>
<td>check two arguments on same dimensionality, extents and coordinates</td>
<td>integer*4 iarg1 (input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 iarg2 (input)</td>
<td>argument number</td>
</tr>
<tr>
<td></td>
<td>If appropriate forward return code ≠ 0 to the function value of simenv_check_user_def_operator()</td>
<td>integer*4 simenv_chk_2args_[f</td>
<td>c] (function value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 0 ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 1 differing dimensionalities</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 2 differing extents</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 3 differing coordinates according to the sub-keyword 'coord_check' in &lt;model&gt;.cfg</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 4 iarg1 = iarg2</td>
</tr>
<tr>
<td>simenv_put_struct_res_[f</td>
<td>c] (inplace, idimens {, iext, ico_nr, ico_beg_pos})</td>
<td>put - potential inplace-storage - dimensionality - extents - formal coordinate number - formal coordinate value begin number of the result</td>
<td>integer*4 inplace (input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= -1 all</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 0 none</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&gt; 0 e.g. = 135 with arguments 1, 3 and 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 idimens (input)</td>
<td>dimensionality of the result</td>
</tr>
<tr>
<td></td>
<td>Currently, only coordinates from the arguments can be assigned to the result.</td>
<td>integer*4 iext(9) (input)</td>
<td>only for idimens &gt; 0: extents of the result</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>iext(1) ... iext(idimens)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 ico_nr(9) (input)</td>
<td>only for idimens &gt; 0: formal coordinate numbers of the result</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ico_nr(1) ... ico_nr(idimens)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 ico_beg_pos(9) (input)</td>
<td>only for idimens &gt; 0: formal coordinate begin position for formal coordinate number ico_nr of the result</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>ico_beg_pos(1) ... ico_beg_pos(idimens)</td>
</tr>
<tr>
<td></td>
<td>Has to be applied in the declarative part and only there.</td>
<td>integer*4 simenv_put_dim_res_[f</td>
<td>c] (function value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 0 ok</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>≠ 0 inconsistency between operands</td>
</tr>
</tbody>
</table>

Tab. 8.16 Operator interface functions to get and put structural information

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. 8.17 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].
### Functions to get and check argument values and to put results in the computational part

<table>
<thead>
<tr>
<th>Function name</th>
<th>Function description</th>
<th>Inputs / outputs / function value</th>
<th>Inputs / outputs / function value</th>
</tr>
</thead>
<tbody>
<tr>
<td>simenv_get_arg_[f</td>
<td>c]</td>
<td>get value of a non-character argument with index index</td>
<td>integer*4 iarg (input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 index (input)</td>
<td>vector index of an argument</td>
</tr>
<tr>
<td></td>
<td></td>
<td>real*4 simenv_get_arg_[f</td>
<td>c] (function value)</td>
</tr>
<tr>
<td>simenv_clip_undef_[f</td>
<td>c]</td>
<td>overflow: set a real<em>8 value to an undefined real</em>4 result if appropriate</td>
<td>real*8 value (input)</td>
</tr>
<tr>
<td></td>
<td>underflow: set a real<em>8 value to real</em>4 0. if appropriate</td>
<td>real*4 simenv_clip_undef_[f</td>
<td>c] (function value)</td>
</tr>
<tr>
<td>simenv_chk_undef_[f</td>
<td>c]</td>
<td>check whether value is undefined before processing it</td>
<td>real*4 value (input)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer*4 simenv_is_undef_[f</td>
<td>c] (function value)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>= 1 value is undefined</td>
</tr>
<tr>
<td>simenv_put_undef_[f</td>
<td>c]</td>
<td>set a result value as undefined</td>
<td>real*4 simenv_put_undef_[f</td>
</tr>
</tbody>
</table>

**Tab. 8.17** Operator interface functions to get / check / put arguments and results

- 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 `<opr>.opr` and has to be located in that directory that is stated in `<model>.cfg` as the hosting directory `opr_directory` for user-defined operators.
- The include file `simenv_opr_f.inc` and `simenv_opr_c.inc` from the SimEnv home directory can be used in user-defined operators to declare the SimEnv operator interface functions for Fortran and/or C/C++ (see also Tab. 10.4).
- Apply the shell script `simenv_opr_[f|c|cpp].lnk <opr>` from the SimEnv home directory to compile and link from `usr_opr_<opr>[f|c|cpp]` an executable `<opr>.opr` that represents the user-defined operator `<opr>`. Like the main program for the operator also
the object $SE_HOME/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/libsimenv.a.

- Tab. 15.13 lists the additionally used symbols when linking a user-defined operator.
- In Section 15.3 on page 157 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 ) (see Tab. 8.17) 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( ) (see Tab. 8.17)
Check usr_opr_matmul_[ f | c ][ f | c ] in Section 15.3 or usr_opr_div.f in the example directory
$SE_HOME/../examples 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 al-
ternatively
- 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 experi-
ment post-processing. In this sense, the definition of a composed operator in SimEnv corresponds with 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 (see 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 targets
- 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 27 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 8.9 Composed operators

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

- Character arguments of operators
  Example: The composed operator error_1 is considered by simenv.chk to be valid though argument 1 of operator count is limited to values ['all' | 'def' | 'undef'] and argument 1 of operator hgr is limited to values ['bin_no' | 'bin_mid']

- Use of "normal" formal arguments in the operator chain with respect to their dimensionality, extents and coordinates
  Example: The composed operator error_2 in is considered by simenv.chk to be valid though the dimensionality of the operator hgr in this constellation is always higher than that of the argument arg1 and consequently, multiplication between arg1 and hgr(.) is impossible.

8.5.4 Operator Description File <model>.odf

<model>.odf is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 8.18. <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 exploited to check a user-defined and/or composed operator syntactically when performing it during experiment post-processing.

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>&lt;nil&gt;</td>
<td>descr</td>
<td>o</td>
<td>any</td>
<td>&lt;string&gt;</td>
<td>general operator descriptions</td>
</tr>
<tr>
<td>opr_defined</td>
<td>&lt;user_defined_operator_name&gt;</td>
<td>descr</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>operator description</td>
</tr>
<tr>
<td></td>
<td>arguments</td>
<td>m</td>
<td></td>
<td></td>
<td>&lt;int_val1&gt;, &lt;int_val2&gt;, &lt;int_val3&gt;, &lt;int_val4&gt;</td>
<td>number of arguments defined for the operator: &lt;int_val1&gt; ≥ 0: character arguments &lt;int_val2&gt; ≥ 0: integer constant arguments &lt;int_val3&gt; ≥ 0: real constant arguments &lt;int_val4&gt; &gt; 0: &quot;normal&quot; arguments</td>
</tr>
</tbody>
</table>
### Tab. 8.18 Elements of an operator description file `<model>.odf`

To Tab. 8.18 the following additional rules and explanations apply:

- For the description of line type check Tab. 11.4 on page 121.
- The sequence of the four integer values `<int_val1>` , ..., `<int_val4>` follows the sequence of arguments in built-in, user-defined and composed operators.
- The sum `<int_val1>` + ... + `<int_val4>` has to be less equal 9.
- Use the SimEnv service `simenv.chk` to check user-defined and composed operators.

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>values</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>opr_composed</td>
<td><code>&lt;composed_operator_name&gt;</code></td>
<td>descr</td>
<td>o</td>
<td>1</td>
<td><code>&lt;string&gt;</code></td>
<td>operator description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>arguments</td>
<td>m</td>
<td>1</td>
<td><code>&lt;int_val1&gt;</code>, <code>&lt;int_val2&gt;</code>, <code>&lt;int_val3&gt;</code>, <code>&lt;int_val4&gt;</code></td>
<td>number of arguments defined for the operator. Explanations and restrictions are the same as for a user-defined operator</td>
</tr>
<tr>
<td></td>
<td></td>
<td>define</td>
<td>m</td>
<td>≥ 1</td>
<td><code>&lt;string&gt;</code></td>
<td>operator definition string Operotor definition can be arranged at a series of define-lines in analogy to the rules for result expressions (see Section 8.1.1).</td>
</tr>
</tbody>
</table>

**Example files:** `world_[f|c|cpp|py|sh].odf`

**Example 8.10 Operator description file `<model>.odf`**
8.6 Undefined Results

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

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

For undefined / nodata value representation check Tab. 10.12.

8.7 Macros and Macro Definition File <model>.mac

- In experiment post-processing a macro is an abbreviation for a result expression, consisting of an operator chain applied on operands.
- Generally, they are model related and they are defined by the user.
- Macros are identified in experiment post-processing expressions by the suffix _m.
- A macro is plugged into a result expression by putting it into parentheses during parsing:

  Example: equ_100yrs_m*test_mac_m
  from Example 8.11 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 (check Tab. 15.10)
  - Integer or real constant arguments of built-in operators (check Tab. 15.11)
  - Character arguments of user-defined operators
  - Operators with respect to dimensionality and dimensions of its operands

In SimEnv macros are defined in the file <model>.mac. <model>.mac is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and values as in Tab. 8.19. <model>.mac describes the user-defined macros.

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>&lt;nil&gt;</td>
<td>descr</td>
<td>o</td>
<td>any</td>
<td>&lt;string&gt;</td>
<td>general macro descriptions</td>
</tr>
<tr>
<td>macro</td>
<td>&lt;macro_name&gt;</td>
<td>descr</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>macro description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unit</td>
<td>m</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>unit of the value of the macro</td>
</tr>
<tr>
<td></td>
<td></td>
<td>define</td>
<td>m</td>
<td>≥ 1</td>
<td>&lt;string&gt;</td>
<td>macro definition string</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>macro definition can be arranged at a series of define-lines in analogy to the rules for result expressions (see Section 8.1.1).</td>
</tr>
</tbody>
</table>

Tab. 8.19 Elements of a macro description file <model>.mac

To Tab. 8.19 the following additional rules and explanations apply:
- For the description of line type check Tab. 11.4 on page 121.
- Values for sub-keywords ‘descr’ and ‘unit’ are not evaluated during parsing a result expression.
general descr Macro definitions for the examples in the SimEnv User Guide

general descr 2nd century tropical level 1 average without

macro equ_100yrs unit 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 | sh ].mac

Example 8.11 User-defined macro definition file <model>.mac

8.8 Wildcard Operands &v& and &t&

In SimEnv, wildcard operands offer a convenient approach to compute a result expression successively for all defined model output variables and experiment targets. 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 variable
&t& wildcard operand for any experiment target

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

Wildcard operands must not be used in macro definitions (see Section 8.7). The wildcard operand &v& for model output variables can not 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).

Keep in mind that the strings &v& and &t& are only substituted in the result string by model variables and/or model targets if they the strings are
- prefixed by [ ( | + | - | / | * | begin of result string] and
- prefixed by [ ( | + | - | / | * | end of result string ]

Having a model output variable definition as in Example 5.1 on page 27 and assuming the experiment description file (b) from Example 6.1 on page 46 then in result-processing

```
results in
behav(' ',sin(&v&))
behav(' ',sin(atom))
behav(' ',sin(bios))
behav(' ',sin(atmo_g))
behav(' ',sin(bios_g))

results in
behav(' ',&v*)&t&)
behav(' ',atmo*p1)
behav(' ',bios*p1)
behav(' ',atmo_g*p1)
behav(' ',bios_g*p1)
behav(' ',atmo*p2)
```


Example 8.12  Experiment post-processing with wildcard operands

8.9  Saving Results

The result files `<model>.res<res_char>.[nc | iee | ascii ]` and `<model>.inf<res_char>. [ iee | ascii ]` contain all the model and experiment information for further processing of results.
9 Visual Experiment Evaluation

Experiment evaluation is based on application of visualization techniques to the output data, computed during experiment post-processing and stored in NetCDF format. Currently, a preliminary version is implemented.

Analysis and evaluation of post-processed data selected and derived from large amount of relevant model output benefits from visualization techniques. Based on metadata information of the post-processed experiment type, the applied operator chain, and the dimensionalities of the post-processor output pre-formed visualization modules are evaluated by a suitability coefficient how they can map the data in an appropriate manner.

The visualization modules offer a high degree of user support and interactivity to cope with multi-dimensional data structures. They cover among others standard techniques such as isolines, isosurfaces, direct volume rendering and a 3D difference visualization techniques (for spatial and temporal data visualization). 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 visualization platform techniques have been designed and implemented, suited in the context of analysis and evaluation of simulated 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 SimEnv.

To get access permission to the SimEnv visualization server use the SimEnv service simenv.key one time. Check Section 10.2 for more information.
## 10 General Control, Services, User Files, and Settings

In a general configuration file `<model>.cfg` the user controls general settings for the simulation environment. 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 settings reflect case sensitivity, nodata values and other information related to SimEnv.

### 10.1 General Configuration File `<model>.cfg`

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>descr</td>
<td>o</td>
<td>any</td>
<td>&lt;string&gt;</td>
<td>general configuration description</td>
<td></td>
</tr>
<tr>
<td></td>
<td>message_level</td>
<td>o</td>
<td>1</td>
<td>[ info</td>
<td>warning</td>
<td>error ]</td>
</tr>
<tr>
<td>model</td>
<td>out_directory</td>
<td>o</td>
<td>1</td>
<td>&lt;directory&gt;</td>
<td>model output directory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>out_format</td>
<td>o</td>
<td>1</td>
<td>[ netcdf</td>
<td>ieee ]</td>
<td>model output format</td>
</tr>
<tr>
<td></td>
<td>out_separation</td>
<td>o</td>
<td>1</td>
<td>[ yes</td>
<td>no ]</td>
<td>indicates whether to store model output in a single file per single run or in one file per experiment</td>
</tr>
<tr>
<td></td>
<td>auto_interface</td>
<td>o</td>
<td>1</td>
<td>[ no</td>
<td>all</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>structure</td>
<td>o</td>
<td>1</td>
<td>[ standard</td>
<td>distributed</td>
<td>parallel ]</td>
</tr>
<tr>
<td>experiment</td>
<td>restart_ini</td>
<td>o</td>
<td>1</td>
<td>[ no</td>
<td>yes ]</td>
<td>perform <code>&lt;model&gt;.ini</code> for experiment re-start</td>
</tr>
<tr>
<td></td>
<td>begin_run</td>
<td>o</td>
<td>1</td>
<td>&lt;int_val&gt;</td>
<td>begin single run number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end_run</td>
<td>o</td>
<td>1</td>
<td>[ last</td>
<td>&lt;int_val&gt; ]</td>
<td>end single run number</td>
</tr>
<tr>
<td></td>
<td>email</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>email notification address</td>
<td></td>
</tr>
<tr>
<td>postproc</td>
<td>out_directory</td>
<td>o</td>
<td>1</td>
<td>&lt;directory&gt;</td>
<td>experiment post-processing output directory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>out_format</td>
<td>o</td>
<td>1</td>
<td>[ netcdf</td>
<td>ieee</td>
<td>ascii ]</td>
</tr>
<tr>
<td></td>
<td>address_default</td>
<td>o</td>
<td>1</td>
<td>[ coordinate</td>
<td>index ]</td>
<td>experiment post-processing address default for model output variables</td>
</tr>
<tr>
<td></td>
<td>coord_check</td>
<td>o</td>
<td>1</td>
<td>[ strong</td>
<td>weak</td>
<td>without ]</td>
</tr>
<tr>
<td></td>
<td>opr_directory</td>
<td>o</td>
<td>1</td>
<td>&lt;directory&gt;</td>
<td>directory the post-processors expects user-defined operator executables</td>
<td></td>
</tr>
<tr>
<td></td>
<td>visualization</td>
<td>o</td>
<td>1</td>
<td>[ yes</td>
<td>no ]</td>
<td>determine whether to directly visualize an entered result during experiment post-processing</td>
</tr>
</tbody>
</table>

**Tab. 10.1** Elements of a general configuration file `<model>.cfg`
In the ASCII file `<model>.cfg` general SimEnv control variables can be declared. `<model>.mdf` is an ASCII file that follows the coding rules in Section 11.1 on page 119 with the keywords, names, sub-keywords, and info as in Tab. 10.1.

To Tab. 10.1 the following additional rules and explanations apply:

- For the description of `line type` check Tab. 11.4 on page 121.
- **For keyword 'general', sub-keyword 'message_level':**
  Message output during simenv.chk and to the model interface log-file `<model>.mlog` is controlled by this information.
  - Specify info to output errors and warnings and additional information
  - Specify warning to output errors and warnings
  - Specify error to output errors
  during simenv.chk and to `<model>.mlog`.
  - **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<run_char>.[ nc | ieee ]`.
- **For keyword 'model', sub-keyword 'auto_interface':**
  Check Section 5.8.
- **For keyword 'model', sub-keyword 'structure':**
  Check Section 5.9.
- **For keyword 'experiment', sub-keyword ['begin_run' | 'end_run']:**
  With the exception of an optimization experiment SimEnv enables to perform an experiment partially by performing only an experiment slice out of the whole run ensemble (see Section 7.5 on page 58). Therefor assign appropriate run numbers to these two descriptors. Make sure that begin and end run represent run number from the experiment (including run number 0) and that begin run ≤ end run. The value string "last" always represents the last simulation run of the whole run ensemble.
  For an optimization experiment these two sub-keywords are ignored.
- **For keyword 'experiment', sub-keyword 'email':**
  After performing an experiment an email is sent to the email address specified in `<string>`.
- **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 'visualization':**
  Specifies whether to directly visualize an entered result during experiment post-processing.

Please keep in mind to ensure consistency of control settings in `<model>.cfg` across different SimEnv services. As an example one has to run experimentation, experiment post-processing and dump with the same value for `out_separation` in `<model>.cfg`.

Tab. 10.2 lists the default values in the general configuration file in the case of absence of the appropriate sub-keyword.
<table>
<thead>
<tr>
<th>keyword</th>
<th>sub-keyword</th>
<th>Default value (*)</th>
<th>For more information see</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>descr</td>
<td>&lt;nil&gt;</td>
<td>above</td>
</tr>
<tr>
<td></td>
<td>message_level</td>
<td>info</td>
<td>above</td>
</tr>
<tr>
<td>model</td>
<td>out_directory</td>
<td>./</td>
<td>above</td>
</tr>
<tr>
<td></td>
<td>out_format</td>
<td>NetCDF</td>
<td>Chapter 12</td>
</tr>
<tr>
<td></td>
<td>out_separation</td>
<td>yes</td>
<td>above</td>
</tr>
<tr>
<td></td>
<td>auto_interface</td>
<td>no</td>
<td>Section 5.8</td>
</tr>
<tr>
<td></td>
<td>structure</td>
<td>standard</td>
<td>Section 5.9 and above</td>
</tr>
<tr>
<td>experiment</td>
<td>restart_ini</td>
<td>no</td>
<td>Section 7.3</td>
</tr>
<tr>
<td></td>
<td>begin_run</td>
<td>0</td>
<td>Section 7.1 - 7.5</td>
</tr>
<tr>
<td></td>
<td>end_run</td>
<td>last</td>
<td>Section 7.1 - 7.5</td>
</tr>
<tr>
<td></td>
<td>email</td>
<td>&lt;nil&gt;</td>
<td>Section 7.1</td>
</tr>
<tr>
<td>postproc</td>
<td>out_directory</td>
<td>./</td>
<td>above</td>
</tr>
<tr>
<td></td>
<td>out_format</td>
<td>NetCDF</td>
<td>Chapter 12</td>
</tr>
<tr>
<td></td>
<td>address_default</td>
<td>coordinate</td>
<td>Section 8.1.3 and above</td>
</tr>
<tr>
<td></td>
<td>coord_check</td>
<td>strong</td>
<td>Section 8.1.5 and above</td>
</tr>
<tr>
<td></td>
<td>opr_directory</td>
<td>./</td>
<td>Section 8.5</td>
</tr>
<tr>
<td></td>
<td>visualization</td>
<td>yes</td>
<td>above</td>
</tr>
</tbody>
</table>

**Tab. 10.2**  
Default values for the general configuration file  
(*): in the case of absence of the appropriate sub-keyword

---

```plaintext
Example 10.1  User-defined general configuration file <model>.cfg
```

---

### 10.2 Main and Auxiliary Services

The following SimEnv service commands are available from the SimEnv home directory $SE_HOME. Besides experiment performance and experiment post-processing there are additional auxiliary SimEnv services to set the SimEnv environment, to check input information consistency, to monitor the status of a running simulation experiment, to dump files of model and experiment post-processor output and to wrap up the SimEnv workspace.
<table>
<thead>
<tr>
<th>SimEnv service</th>
<th>Use to</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Main Services</strong></td>
<td></td>
</tr>
<tr>
<td>simenv.run &lt;model&gt;</td>
<td>prepare and run an experiment (see Section 7.1)</td>
</tr>
<tr>
<td>simenv.rst &lt;model&gt;</td>
<td>restart an experiment (see Section 7.3)</td>
</tr>
<tr>
<td>simenv.res &lt;model&gt; {{ new</td>
<td>append</td>
</tr>
<tr>
<td>simenv.vis &lt;model&gt; {{ latest</td>
<td>&lt;res&gt; }}</td>
</tr>
<tr>
<td><strong>Auxiliary Services</strong></td>
<td></td>
</tr>
<tr>
<td>simenv.chk &lt;model&gt;</td>
<td>check on model script files (&lt;model&gt;.run, &lt;model&gt;.rst, &lt;model&gt;.ini, &lt;model&gt;.end) check &lt;model&gt;.cfg &lt;model&gt;.edf &lt;model&gt;.odf &lt;model&gt;.gdf &lt;model&gt;.mdf &lt;model&gt;.mac generate pre-experiment output statistics</td>
</tr>
<tr>
<td>simenv.sts &lt;model&gt; {{ &lt;sleep&gt; }}</td>
<td>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.</td>
</tr>
<tr>
<td>simenv.dmp &lt;model&gt; &lt;dmp_modus&gt;</td>
<td>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 &lt;model&gt;.cfg. Model output variables and post-processor results in IEEE and/or ASCII format with a dimensionality greater than 1 are listed according to Fortran storage model for multi-dimensional fields (see Section 15.7 - Glossary).</td>
</tr>
<tr>
<td>simenv.cpl &lt;model&gt; {{ &lt;run&gt; } { &lt;file&gt; }}</td>
<td>complete sequence of SimEnv services simenv.chk, simenv.run, simenv.res, simenv.vis, simenv.dmp simenv.res is performed with input file &lt;file&gt; (if available) and interactively, for both optionally only for single run &lt;run&gt;.</td>
</tr>
<tr>
<td>simenv.cln &lt;model&gt;</td>
<td>clean up model and experiment post-processor output files Deletes all model output files, post-processor output files, log-files, and auxiliary files of a model according to the settings in &lt;model&gt;.cfg</td>
</tr>
<tr>
<td>simenv.cpy &lt;model&gt;</td>
<td>copy all SimEnv example files &lt;model&gt;* from the example directory $SE_HOME/../examples to the current directory. Additionally, example files of user-defined operators and for models world_ [ f</td>
</tr>
<tr>
<td>simenv.hlp &lt;topics&gt;</td>
<td>acquire basic SimEnv help information for the specified topics</td>
</tr>
<tr>
<td>simenv.key &lt;user_name&gt;</td>
<td>generate a ssh(2)-key to get password-free access to the visualization server. Start this service only one time before the first access to simenv.vis and/or simenv.res or if the ssh(2)-key does not work properly. An email from SimEnv will be sent from SimEnv when the password-free server access is possible.</td>
</tr>
</tbody>
</table>

Tab. 10.3 SimEnv services
With the exception of the simenv.cpy, simenv.hlp and simenv.key:
Start a service only from the current workspace.
With the exception of simenv.sts:
Do not start a SimEnv service from a workspace where an other SimEnv service is still active.

10.3 SimEnv and User Include Files and Link Scripts

In Tab. 10.4 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.

<table>
<thead>
<tr>
<th>File / location</th>
<th>Used in / generated</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>simenv_mod_ [f</td>
<td>c</td>
<td>cpp].lnk</td>
</tr>
<tr>
<td>simenv_opr_ [f</td>
<td>c</td>
<td>cpp].lnk</td>
</tr>
<tr>
<td>simenv_mod_ [f</td>
<td>c].inc</td>
<td>used in: interfaced Fortran/C/C++ models</td>
</tr>
<tr>
<td>simenv_opr_ [f</td>
<td>c].inc</td>
<td>used in: interfaced Fortran/C/C++ models</td>
</tr>
<tr>
<td>&lt;model&gt;.lnk</td>
<td>generated: by the user used in: experiment preparation (only run, not re-start, (only if auto_interlace ≠ no in &lt;model&gt;.cfg) and stand alone</td>
<td>shell script to link an interfaced Fortran/C/C++ model. Used in the course of experiment preparation for experiment run (not re-start) if a semi-automated model interface was declared in &lt;model&gt;.cfg for the appropriate programming languages. Can also be used stand alone for non-semi-automated model interface. Is normally based on $SE_HOME/simenv_mod_[f</td>
</tr>
<tr>
<td>&lt;model&gt;_[f</td>
<td>c</td>
<td>py</td>
</tr>
</tbody>
</table>

Tab. 10.4  SimEnv and user include files and link scripts
(*) make sure by the Unix / Linux command chmod u+x <file> that a file <file> has execute permission.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Data type</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>simenv_sts</td>
<td>integer*4 / int</td>
<td>SimEnv interface function value</td>
</tr>
<tr>
<td>simenv_run_int</td>
<td>integer*4 / int</td>
<td>single run number</td>
</tr>
<tr>
<td>simenv_zero</td>
<td>real*4 / float</td>
<td>auxiliary variable, set to 0.</td>
</tr>
</tbody>
</table>

Tab. 10.5  Contents of $SE_HOME/simenv_mod_[f|c].inc
(without definition of interface functions)
### 10.4 User Shell Scripts and Files

<table>
<thead>
<tr>
<th>Shell script / file (in the current workspace $SE_WS)</th>
<th>Explanation</th>
<th>Exist status</th>
<th>For more information see Section</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;model&gt;.cfg</code></td>
<td>ASCII user-defined general configuration file</td>
<td>optional</td>
<td>10.1</td>
</tr>
<tr>
<td><code>&lt;model&gt;.mdf</code></td>
<td>ASCII user-defined model (variables) description file</td>
<td>mandatory</td>
<td>5.1</td>
</tr>
<tr>
<td><code>&lt;model&gt;.edf</code></td>
<td>ASCII user-defined experiment description file</td>
<td>mandatory</td>
<td>6.1</td>
</tr>
<tr>
<td><code>&lt;model&gt;.mac</code></td>
<td>ASCII user-defined macro description file</td>
<td>optional</td>
<td>8.7</td>
</tr>
<tr>
<td><code>&lt;model&gt;.odf</code></td>
<td>ASCII user-defined operator description file</td>
<td>optional</td>
<td>8.5.4</td>
</tr>
<tr>
<td><code>&lt;model&gt;.gdf</code></td>
<td>ASCII user-defined GAMS model output description file</td>
<td>for GAMS models mandatory</td>
<td>5.7.2</td>
</tr>
<tr>
<td><code>&lt;model&gt;.run (*)</code></td>
<td>model shell script to wrap the model executable</td>
<td>mandatory</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;model&gt;.rst (*)</code></td>
<td>model shell script to prepare single model run restart</td>
<td>optional</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;model&gt;.ini (*)</code></td>
<td>model shell script to prepare simulation experiment additionally to standard SimEnv preparation</td>
<td>optional, for Python and GAMS models mandatory and standardized</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;model&gt;.end (*)</code></td>
<td>model shell script to clean up simulation experiment</td>
<td>optional, for GAMS models mandatory and standardized</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;model&gt;.link (*)</code></td>
<td>model shell script to link an interfaced C/C++/Fortran model. Used in the course of experiment preparation for experiment run (not re-start) if a semi-automated model interface was declared in <code>&lt;model&gt;.cfg</code> for the appropriate programming languages</td>
<td>optional</td>
<td>5.8</td>
</tr>
<tr>
<td><code>&lt;model&gt;.jcf_dis</code></td>
<td>user-specific job control file to submit a job by the load leveler in distributed mode</td>
<td>optional</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;model&gt;.jcf_par</code></td>
<td>user-specific job control file to submit a job by the load leveler in parallel mode</td>
<td>optional</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;model&gt;.jcf_seq</code></td>
<td>user-specific job control file to submit a job by the load leveler in sequential mode</td>
<td>optional</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;model&gt;.opt_opt</code></td>
<td>user-specific control and option file for experiment type optimization</td>
<td>optional</td>
<td>6.5.1</td>
</tr>
<tr>
<td><code>&lt;model&gt;_&lt;run_char&gt;.err</code></td>
<td>touch / create this file in the model or in <code>&lt;model&gt;.run</code> as an indicator to stop the complete experiment after <code>&lt;model&gt;.run</code> has been finished for the single model run <code>&lt;run_char&gt;</code></td>
<td>optional</td>
<td>7.6</td>
</tr>
<tr>
<td><code>&lt;opr&gt;.opr</code> (*)</td>
<td>executable for user-defined operator <code>&lt;opr&gt;</code></td>
<td>optional</td>
<td>8.5</td>
</tr>
</tbody>
</table>

*User files and shell scripts to perform any SimEnv service

(*) make sure by the Unix / Linux command `chmod u+x <file>`
that a file `<file>` has execute permission*
<table>
<thead>
<tr>
<th>File / location</th>
<th>Generated in</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Permanent files</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;model&gt;.edf_adj</code></td>
<td>experiment preparation (all but optimization)</td>
<td>ASCII adjustment input file for the run ensemble derived from <code>&lt;model&gt;.edf</code> Record no. n+1 corresponds to single run no. n. Column no. m of each record is the adjustment for experiment target no. m in the edf-file</td>
</tr>
<tr>
<td>`&lt;model&gt;_[ f</td>
<td>c</td>
<td>py</td>
</tr>
<tr>
<td>`&lt;model&gt;.out&lt;run_char&gt; [. nc</td>
<td>ieee ]`</td>
<td>experiment performance (if out_separation = yes in <code>&lt;model&gt;.cfg</code>)</td>
</tr>
<tr>
<td>`&lt;model&gt;.outtail [. nc</td>
<td>ieee ]`</td>
<td>experiment performance (if out_separation = no in <code>&lt;model&gt;.cfg</code>)</td>
</tr>
<tr>
<td><code>&lt;model&gt;.elog</code></td>
<td>experiment performance</td>
<td>ASCII minutes file of experiment performance (simenv.run and all successive simenv.rst)</td>
</tr>
<tr>
<td><code>&lt;model&gt;.mlog</code></td>
<td>experiment performance</td>
<td>ASCII minutes file of model interface functions performance (simenv.run and all successive simenv.rst) <code>&lt;model&gt;.mlog</code> is organized single run by single run</td>
</tr>
<tr>
<td><code>&lt;model&gt;.nlog</code></td>
<td>experiment performance</td>
<td>ASCII minutes file of native - model specific experim. prepar. by <code>&lt;model&gt;.ini</code> - single runs model output by <code>&lt;model&gt;.run</code> - single run restart preparation by <code>&lt;model&gt;.rst</code> - model specific experim. wrap-up by <code>&lt;model&gt;.end</code> performances, redirected from terminal (simenv.run and all successive simenv.rst) <code>&lt;model&gt;.nlog</code> is organized single run by single run</td>
</tr>
<tr>
<td>`&lt;model&gt;.res&lt;res_char&gt; [. nc</td>
<td>ieee</td>
<td>ascii ]`</td>
</tr>
<tr>
<td>`&lt;model&gt;.inf&lt;res_char&gt; [. ieee</td>
<td>ascii ]`</td>
<td>experiment post-processing</td>
</tr>
<tr>
<td><code>run&lt;run_char&gt;</code></td>
<td>experiment performance (only for GAMS models)</td>
<td>sub-directory for GAMS model performance that are kept according to the sub-keyword ‘keep_runs’ in <code>&lt;model&gt;.gdf</code></td>
</tr>
<tr>
<td><code>&lt;model&gt;.olog</code></td>
<td>experiment performance (only for experiment type optimization)</td>
<td>ASCII minutes file of optimization experiment performance</td>
</tr>
<tr>
<td><code>&lt;model&gt;.edf_cf</code></td>
<td>experiment performance (only for experiment type optimization)</td>
<td>ASCII file of cost function values. Record no. n+1 corresponds to single run no. n.</td>
</tr>
</tbody>
</table>
### File / location

<table>
<thead>
<tr>
<th>Temporary files</th>
</tr>
</thead>
<tbody>
<tr>
<td>(do not delete during performing the corresponding service)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>File</th>
<th>Generated in</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>`&lt;model&gt;.[ cfg</td>
<td>mdf</td>
<td>edf</td>
</tr>
<tr>
<td><code>$SE_WS</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;model&gt;.out&lt;run_char&gt;.[ nc</td>
<td>ieee ]`</td>
<td>experiment performance (if out_separation = yes in <code>&lt;model&gt;.cfg</code>)</td>
</tr>
<tr>
<td><code>model.out_directory</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>&lt;model&gt;.res00.nc</code></td>
<td>experiment post-processing</td>
<td>NetCDF representation of the current result for visualization during experiment post-processing (only for value “yes” of sub-keyword ‘visualization’ in <code>&lt;model&gt;.cfg</code>)</td>
</tr>
<tr>
<td><code>$SE_WS</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>asa_opt</code></td>
<td>experiment performance (only for experiment type optimization)</td>
<td>auxiliary files for experiment type optimization</td>
</tr>
<tr>
<td><code>asa_out</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>asa_usr_out</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>$SE_WS</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>run&lt;run_char&gt;</code></td>
<td>experiment performance (only for GAMS models)</td>
<td>sub-directory for GAMS model performance that are not kept according to the sub-keyword ‘keep_runs’ in <code>&lt;model&gt;.gdf</code></td>
</tr>
<tr>
<td><code>sub-direct. of $SE_WS</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>`&lt;model&gt;._[ pre</td>
<td>main</td>
<td>post ].inc`</td>
</tr>
<tr>
<td><code>$SE_WS</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>simenv_get_experiment.exe</code></td>
<td>experiment post-processing</td>
<td>auxiliary file for operator <code>get_experiment</code></td>
</tr>
<tr>
<td><code>$SE_WS</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>simenv_*tmp</code></td>
<td>all services</td>
<td>auxiliary files</td>
</tr>
<tr>
<td><code>$SE_WS</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 10.7** Files generated during performance of SimEnv services

For the current workspace `$SE_WS` see Tab. 10.13.

Fig. 10.1 sketches usage of main SimEnv user shell scripts and files in the course of model interfacing, experiment preparation and performance, experiment post-processing, and visual evaluation of post-processed results.
10.5 Built-In Names

SimEnv has a number of built-in model output variable, coordinate and shell script variable names that can not be used for corresponding user-defined names.

Tab. 10.8 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.

<table>
<thead>
<tr>
<th>Built-in model output variable name</th>
<th>Dimensionality</th>
<th>Extents</th>
<th>Data type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sim_time</td>
<td>0</td>
<td></td>
<td>float</td>
<td>elapsed simulation time in seconds (rounded to two decimal places) when performing &lt;model&gt;.run</td>
</tr>
</tbody>
</table>

**Tab. 10.8** Built-in model output variables
Tab. 10.9 lists the built-in (pre-defined) coordinates that are used in experiment post-processing when additional dimensions are generated by an operator.

<table>
<thead>
<tr>
<th>Built-in coordinate name</th>
<th>Generated by operator</th>
<th>Meaning</th>
<th>Definition (check Tab. 11.6)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin_mid</td>
<td>hgr, hgr_e, hgr_l</td>
<td>bin mid values</td>
<td>equidist_end &lt;xx&gt;&lt;yy&gt; 9999999 with &lt;xx&gt; = first bin mid &lt;yy&gt; = bin width</td>
</tr>
<tr>
<td>bin_no</td>
<td>hgr, hgr_e, hgr_l</td>
<td>bin numbers</td>
<td>equidist_end 1(1)9999999</td>
</tr>
<tr>
<td>incr</td>
<td>lin_abs, lin_rel, sens_abs, sens_rel, sym_abs, sym_rel</td>
<td>increment values</td>
<td>dependent on experiment description and operator arguments</td>
</tr>
<tr>
<td>index</td>
<td>maxprop, maxprop_l, minprop, minprop_l</td>
<td>index number</td>
<td>equidist_end 1(1)9999999</td>
</tr>
<tr>
<td>run</td>
<td>ens</td>
<td>run numbers</td>
<td>equidist_end 1(1)9999999</td>
</tr>
<tr>
<td>sign</td>
<td>sens_abs, sens_rel</td>
<td>signs of incremental change: -1: -ε +1: +ε</td>
<td>equidist_end -1(2)1</td>
</tr>
<tr>
<td>stat_measure</td>
<td>stat_full, stat_red</td>
<td>basic statistical measures: 1: Deterministic case 2: Minimum 3: Maximum 4: Mean 5: Variance 6: Positive distance of confidence measure 1 from mean 7: Positive distance of confidence measure 2 from mean 8: Median 9: Quantile of quantile value 1 10: Quantile of quantile value 2</td>
<td>equidist_end 1(1)10</td>
</tr>
<tr>
<td>target_sequ</td>
<td>lin_abs, lin_rel, sens_abs, sens_rel, sym_abs, sym_rel</td>
<td>sequence of targets: 1: 1st target in edf-file 2: 2nd target in edf-file ...</td>
<td>equidist_end 1(1)9999999</td>
</tr>
<tr>
<td>&lt;target_name&gt;</td>
<td>behav</td>
<td>target values</td>
<td>dependent on experiment description and operator arguments</td>
</tr>
</tbody>
</table>

Tab. 10.9  
Built-in coordinates

Tab. 10.10 lists the built-in (pre-defined) shell script variables that are defined / used from the model coupling interface dot scripts $SSE_HOME/simenv_*_sh and that are finally available in <model>.run.

<table>
<thead>
<tr>
<th>Built-in shell script variable name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>run_int</td>
<td>current run number as integer</td>
</tr>
<tr>
<td>run_char</td>
<td>current run number as character string</td>
</tr>
<tr>
<td>target_name</td>
<td>target name for simenv_get_sh</td>
</tr>
<tr>
<td>target_def_val</td>
<td>default target value for simenv_get_sh</td>
</tr>
</tbody>
</table>

Tab. 10.10  
Built-in shell script variables in <model>.run
10.6 Case Sensitivity

As stated in Tab. 10.11 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.

<table>
<thead>
<tr>
<th>Where?</th>
<th>Entity</th>
<th>Case sensitivity</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>overall</td>
<td>• model name</td>
<td>sensitive</td>
<td>simenv.chk World_f</td>
</tr>
<tr>
<td>user-defined files</td>
<td>• keyword</td>
<td>insensitive</td>
<td>experiment END_RUN last</td>
</tr>
<tr>
<td>(see Section 11.1)</td>
<td>• name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>exception: GAMS model file name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in &lt;model&gt;.gdf</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• sub-keyword</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• information &lt;value&gt; exception:</td>
<td>insensitive</td>
<td>experiment end_run LAST</td>
</tr>
<tr>
<td></td>
<td>• &lt;directory&gt; and &lt;file_name&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- for &lt;sub-keyword&gt; =</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>'string&gt;_directory'</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>- and in &lt;val_list&gt;</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• &lt;value&gt; for &lt;sub-keyword&gt; =</td>
<td>insensitive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>['descr'</td>
<td>'unit'])</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• model interface</td>
<td>insensitive</td>
<td>call simenv_put_f('ATMO', atmo)</td>
</tr>
<tr>
<td></td>
<td>• variable and target name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• optional result description</td>
<td>sensitive</td>
<td></td>
</tr>
<tr>
<td></td>
<td>and unit</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• variable and target name</td>
<td>insensitive</td>
<td>3e-6<em>exp(atmo) + 3E-6</em>EXP(ATMO)</td>
</tr>
<tr>
<td></td>
<td>• operator name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• number</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• macro name</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• macro identifier _m</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• character arguments of built-in operators with pre-defined values (check Tab. 15.10)</td>
<td>insensitive</td>
<td>count('ALL', atmo)</td>
</tr>
<tr>
<td></td>
<td>• character arguments of built-in operators without pre-defined values (check Tab. 15.10)</td>
<td>sensitive</td>
<td>get_table_fct('MyFile.dat', atmo) get_experiment('./', 'Model_f', '', atmo)</td>
</tr>
<tr>
<td></td>
<td>• character arguments of user-defined operators</td>
<td>sensitive</td>
<td>char_test('arg11', 'Arg21', atmo)</td>
</tr>
</tbody>
</table>

| Tab. 10.11          | Case sensitivity of SimEnv entities |
10.7 Nodata Representation

For model output with the SimEnv model coupling interface functions and for experiment post-processor output the following data type specific nodata values are used to represent undefined (unwritten) model output or undefined post-processor output:

<table>
<thead>
<tr>
<th>Data type</th>
<th>Nodata value</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>= 127</td>
</tr>
<tr>
<td>short</td>
<td>= 32767</td>
</tr>
<tr>
<td>int</td>
<td>= 2147483648</td>
</tr>
<tr>
<td>float</td>
<td>≥ 3.4E+38</td>
</tr>
<tr>
<td>double</td>
<td>≥ 1.79D+308</td>
</tr>
</tbody>
</table>

Tab. 10.12 Data type related nodata values

10.8 Environment Variables

The following operating system environment variables are used by SimEnv. Additionally, make sure that in the shell the noclobber option is not set.

<table>
<thead>
<tr>
<th>Environment variable</th>
<th>Meaning</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISPLAY</td>
<td>machine / screen that the X11-system uses for displaying windows</td>
<td>has to be defined by the user only for visualization matters in SimEnv services simenv.res and simenv.vis: Value = machine dependent Specify also explicitly when logged in at a machine by using a secure socket shell client ssh(2)</td>
</tr>
<tr>
<td>SE_HOME</td>
<td>SimEnv home directory</td>
<td>Value = /usr/local/simenv/bin Prefix PATH by $SE_HOME to get any SimEnv service directly</td>
</tr>
<tr>
<td>PYTHONPATH</td>
<td>path to search Python files</td>
<td>Value = dependent on Python installation Expanded by $SE_HOME</td>
</tr>
<tr>
<td>PYTHON_VERSION</td>
<td>Python version</td>
<td>Value = dependent on Python installation</td>
</tr>
<tr>
<td>PYTHON_ROOT</td>
<td>Python root directory</td>
<td>Value = dependent on Python installation</td>
</tr>
<tr>
<td>SE_GUI</td>
<td>identificator for GUI / non-GUI version</td>
<td>defined automatically in any SimEnv service Value = [ yes</td>
</tr>
<tr>
<td>SE_MOD</td>
<td>model name</td>
<td>defined automatically in any SimEnv service Value = &lt;model&gt;</td>
</tr>
<tr>
<td>SE_OS</td>
<td>operating system specification</td>
<td>defined automatically in any SimEnv service Value = [ AIX</td>
</tr>
<tr>
<td>SE_WS</td>
<td>current SimEnv workspace</td>
<td>defined automatically in any SimEnv service Value = &lt;directory&gt;</td>
</tr>
<tr>
<td>SE_RUN</td>
<td>run number of a single run</td>
<td>defined automatically in &lt;model&gt;.run and &lt;model&gt;.rst Value = &lt;run_int&gt;</td>
</tr>
<tr>
<td>SE_RUN1</td>
<td>first single run of an experiment</td>
<td>defined automatically in &lt;model&gt;.run and &lt;model&gt;.rst Value = [ yes</td>
</tr>
</tbody>
</table>

Tab. 10.13 Environment variables
11 Structure of User-Defined Files, Coordinate Transformation Files, Value Lists

Basic information to describe general control settings of SimEnv, model output variables, the experiment itself, macros and user-defined operators as well as GAMS model specific information is stored in user-defined files. They are ASCII files and have a common structure that is described in this chapter. Additionally, coordinate transformation files are described and value lists are defined in general.

11.1 General Structure of User-Defined Files

All user-defined files listed in Tab. 11.1 have the same structure. They are ASCII-files with the following record structure:

{ <sep> } <keyword> <sep> { <name> <sep> } <sub-keyword> <sep> <value> { <sep> }

with

- <name> is the name of a
  - model output variable
  - GAMS model source file
  - experiment target
  - 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 below. Sequence of names in the separated name spaces (name spaces of coordinates, model output variables, experiment targets, user-defined operators, macros) during processing is determined by the sequence the name occur the first time in the appropriate user file.

Lines consisting only from separator characters as well as lines starting with a # as the first non-separator character are handled as comment lines. For case sensitivity of the contents of user-defined files check Tab. 10.11 on page 117.

<table>
<thead>
<tr>
<th>File</th>
<th>Contents</th>
<th>See description in Section</th>
<th>See description on page</th>
</tr>
</thead>
<tbody>
<tr>
<td>model.cfg</td>
<td>general configuration file</td>
<td>10.1</td>
<td>107</td>
</tr>
<tr>
<td>model.mdf</td>
<td>model output description file</td>
<td>5.1</td>
<td>21</td>
</tr>
<tr>
<td>model.gdf</td>
<td>GAMS description file</td>
<td>5.7.2</td>
<td>36</td>
</tr>
<tr>
<td>model.edf</td>
<td>experiment description file</td>
<td>6.1</td>
<td>43</td>
</tr>
<tr>
<td>model.odf</td>
<td>operator description file</td>
<td>8.5.4</td>
<td>99</td>
</tr>
<tr>
<td>model.mac</td>
<td>macro description file</td>
<td>8.7</td>
<td>101</td>
</tr>
<tr>
<td>arbitrary file name</td>
<td>coordinate transformation file</td>
<td>11.2</td>
<td>122</td>
</tr>
</tbody>
</table>

Tab. 11.1 User-defined files with general structure
The following restrictions hold for user-defined files:

<table>
<thead>
<tr>
<th>Element</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>line length</td>
<td>max. 160 characters</td>
</tr>
<tr>
<td>&lt;name&gt;</td>
<td>max. 20 characters</td>
</tr>
<tr>
<td>(*) first character has to be a letter</td>
<td></td>
</tr>
<tr>
<td>(*) must not end on _m</td>
<td></td>
</tr>
<tr>
<td>(*) must not contain elemental operators and characters . and : (check Tab. 8.3 on page 70)</td>
<td></td>
</tr>
<tr>
<td>&lt;value&gt;</td>
<td>for sub-keyword = ‘descr’ without &lt;name&gt;: max. 512 characters (total sum over all lines)</td>
</tr>
<tr>
<td></td>
<td>for sub-keyword = ‘descr’ with &lt;name&gt;: max. 128 characters</td>
</tr>
<tr>
<td></td>
<td>for sub-keyword = ‘&lt;string&gt;_directory’: max. 100 characters (for the resulting resolved directory string, directory can contain operating system environment variables)</td>
</tr>
<tr>
<td></td>
<td>for sub-keyword = ‘unit’: max. 32 characters</td>
</tr>
</tbody>
</table>

**Tab. 11.2** Constraints in user-defined files

(*) with the exception for GAMS model source code file names

Tab. 11.3 lists the reserved (forbidden) names, file names and directories to files that can not be declared in user-defined files.

<table>
<thead>
<tr>
<th>Element</th>
<th>Reserved (forbidden) names</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;name&gt; excepted for GAMS model source code file names</td>
<td>built-in model output variables according to Tab. 10.8</td>
</tr>
<tr>
<td></td>
<td>built-in coordinates according to Tab. 10.9</td>
</tr>
<tr>
<td></td>
<td>special keywords in &lt;model&gt;.edf for behavioural analysis: [ default</td>
</tr>
<tr>
<td>&lt;directory&gt;</td>
<td>can contain operating system environment variables ($...)</td>
</tr>
<tr>
<td></td>
<td>If &lt;directory&gt; is specified in a relative manner it relates to the current workspace.</td>
</tr>
<tr>
<td>&lt;file_name&gt;</td>
<td>SimEnv file names according to Tab. 10.6 and Tab. 10.7</td>
</tr>
</tbody>
</table>

**Tab. 11.3** Reserved names and file names in user-defined files
The line type in the description table for a user-defined file specifies whether a keyword / sub-keyword combination can be omitted.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>User-defined file</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>m</td>
<td>all files</td>
<td>mandatory</td>
</tr>
<tr>
<td>o</td>
<td>all files</td>
<td>optional</td>
</tr>
<tr>
<td>c1</td>
<td>&lt;model&gt;.mdf</td>
<td>conditional 1: forbidden for variables with dimensionality = 0 manditory for variables with dimensionality &gt; 0</td>
</tr>
<tr>
<td></td>
<td>keyword 'variable'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub-keyword ['coords'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>'index_extents']</td>
<td></td>
</tr>
<tr>
<td>c2</td>
<td>&lt;model&gt;.mdf</td>
<td>conditional 2: forbidden for variables with dimensionality = 0 optional for variables with dimensionality &gt; 0</td>
</tr>
<tr>
<td></td>
<td>keyword 'variable'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub-keyword 'coord_extents'</td>
<td></td>
</tr>
<tr>
<td>c3</td>
<td>&lt;model&gt;.edf</td>
<td>conditional 3: mandatory for experiment type = Monte Carlo analysis forbidden for experiment type = local sensitivity analysis conditional for experiment type = behavioural analysis</td>
</tr>
<tr>
<td></td>
<td>keyword 'target'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub-keyword 'adjusts'</td>
<td></td>
</tr>
<tr>
<td>c4</td>
<td>&lt;model&gt;.edf</td>
<td>conditional 4: for adjusts = distr ... forbidden for adjusts = file ...</td>
</tr>
<tr>
<td></td>
<td>for Monte Carlo analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>keyword 'target'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub-keyword 'sampling'</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>&lt;model&gt;.edf</td>
<td>alternatively: either mandatory for all experiment targets or forbidden for all experiment targets</td>
</tr>
<tr>
<td></td>
<td>for behavioural analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>keyword 'target'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub-keyword 'adjusts'</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>&lt;model&gt;.edf</td>
<td>forbidden</td>
</tr>
<tr>
<td></td>
<td>for local sensitivity analysis</td>
<td></td>
</tr>
<tr>
<td></td>
<td>keyword 'target'</td>
<td></td>
</tr>
<tr>
<td></td>
<td>sub-keyword 'adjusts'</td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 11.4** Line types in user-defined files

```
mac descr This is a macro description file
mac descr for the SimEnv User Guide
macro pol_atmo descr atmo outside polar reg., final time, level 1
macro pol_atmo unit without
macro pol_atmo define atmo(c=84:-56,*,c=1,c=20)
macro m1 define avg(atmo_g(c=11:20))
...
```

**Example 11.1** Structure of a user-defined file
11.2 Coordinate Transformation File

Some operators (currently, `get_experiment` and `get_data`) enable access to external data. Most of these operators derive in general 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 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 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 (see Section 10.1).

<table>
<thead>
<tr>
<th>keyword</th>
<th>name</th>
<th>sub-keyword</th>
<th>Line type</th>
<th>Max. line nmb.</th>
<th>value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>general</td>
<td>descr</td>
<td>o</td>
<td>any</td>
<td>&lt;string&gt;</td>
<td>general transformation description</td>
<td></td>
</tr>
<tr>
<td>modify</td>
<td>&lt;original_coordinate_name&gt;</td>
<td>rename</td>
<td>o</td>
<td>1</td>
<td>&lt;new_name&gt;</td>
<td>renames original coordinate</td>
</tr>
<tr>
<td></td>
<td>position_shift</td>
<td>o</td>
<td>1</td>
<td>&lt;real_val&gt;</td>
<td>shifts all values of the original coordinate by the specified value &lt;position_shift_val&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>values_shift</td>
<td>o</td>
<td>1</td>
<td>&lt;int_val&gt;</td>
<td>shifts the result values on the original coordinate by the specified positions &lt;values_shift_val&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>values_add</td>
<td>o</td>
<td>1</td>
<td>&lt;val_list&gt;</td>
<td>defines &lt;values_shift_val&gt; values to add to the coordinate values (for syntax see Tab. 11.6)</td>
<td></td>
</tr>
<tr>
<td>assign</td>
<td>[&lt;original_coordinate_name&gt;</td>
<td>coord</td>
<td>o</td>
<td>1</td>
<td>&lt;co_name&gt;</td>
<td>assign to the dimension with coordinate number &lt;coordinate_nmb&gt; (only for operator <code>get_data('ascii',...)</code> and/or &lt;original_coordinate_name&gt; (else) an already defined coordinate or a coordinate defined by the keyword ‘coordinate’</td>
</tr>
<tr>
<td></td>
<td>]</td>
<td></td>
<td></td>
<td></td>
<td>&lt;co_val1&gt;,&lt;co_val2&gt;</td>
<td>assigns start and end coordinate value to the dimension of the result under consideration</td>
</tr>
<tr>
<td>coordinate</td>
<td>&lt;new_coordinate_name&gt;</td>
<td>descr</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>coordinate axis description</td>
</tr>
<tr>
<td></td>
<td></td>
<td>unit</td>
<td>o</td>
<td>1</td>
<td>&lt;string&gt;</td>
<td>coordinate axis unit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>values</td>
<td>o</td>
<td>1</td>
<td>&lt;val_list&gt;</td>
<td>strictly monotonic sequence of coordinate values (for syntax see Tab. 11.6)</td>
</tr>
</tbody>
</table>

**Tab. 11.5** Elements of a coordinate transformation file
To Tab. 11.5 the following additional rules and explanations apply:

- For the description of line type check Tab. 11.4 on page 121.
- With the sub-keyword 'values_shift' result values can be shifted on the corresponding coordinate by \(<values\text{\_shift_val}> coordinate values. Consequently, \(<values\text{\_shift_val}> coordinate values have to be appended at the end of the coordinate for a positive value of \(<values\text{\_shift_val}> and/or have to be inserted at the begin of the coordinate for a negative value of \(<values\text{\_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'.

- Coordinate numbers \(<coordinate\text{\_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 can not be checked by the SimEnv service simenv.chk or when starting the service simenv.res.

<table>
<thead>
<tr>
<th>dimension</th>
<th>coordinate name</th>
<th>coordinate definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>dim1</td>
<td>list 1,10,100,1000</td>
</tr>
<tr>
<td>2</td>
<td>dim2</td>
<td>equidist_end 2(2)20</td>
</tr>
<tr>
<td>3</td>
<td>dim3</td>
<td>equidist_end 3(3)30</td>
</tr>
<tr>
<td>4</td>
<td>dim4</td>
<td>equidist_end 4(1)43</td>
</tr>
<tr>
<td>5</td>
<td>dim5</td>
<td>equidist_end 5(1)50</td>
</tr>
</tbody>
</table>

Further, assume the coordinate transformation file model.trf as

<table>
<thead>
<tr>
<th>general</th>
<th>descr</th>
<th>example of a coordinate transformation file</th>
</tr>
</thead>
<tbody>
<tr>
<td>modify</td>
<td>dim1</td>
<td>rename new1</td>
</tr>
<tr>
<td>modify</td>
<td>dim1</td>
<td>position_shift 3.</td>
</tr>
<tr>
<td>modify</td>
<td>dim1</td>
<td>values_shift +2</td>
</tr>
<tr>
<td>modify</td>
<td>dim1</td>
<td>values_add list 1006,1009</td>
</tr>
<tr>
<td>modify</td>
<td>dim3</td>
<td>values_shift -3</td>
</tr>
<tr>
<td>assign</td>
<td>dim4</td>
<td>coord lat</td>
</tr>
<tr>
<td>assign</td>
<td>dim4</td>
<td>coord_extent 88.:.-68.</td>
</tr>
<tr>
<td>assign</td>
<td>dim5</td>
<td>coord new2</td>
</tr>
<tr>
<td>assign</td>
<td>dim5</td>
<td>coord_extent 50.:5.</td>
</tr>
</tbody>
</table>
coordinate new2 descr new coordinate
countordinate new2 values equidist_end 50(-1)5

In experiment post-processing the result of the expression

get_experiment('mydir', 'model', 'model.trf', modvar1+modvar2)

is a 5-dimensional data structure with

<table>
<thead>
<tr>
<th>dimension</th>
<th>coordinate name</th>
<th>definition</th>
<th>coordinate use</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>new1</td>
<td>list 103,1003,1006,1009</td>
<td>= coordinate definition</td>
</tr>
<tr>
<td>2</td>
<td>dim2</td>
<td>equidist_end 2(2)20</td>
<td>= coordinate definition</td>
</tr>
<tr>
<td>3</td>
<td>dim3</td>
<td>equidist_end -6(3)21</td>
<td>= coordinate definition</td>
</tr>
<tr>
<td>4</td>
<td>lat</td>
<td>equidist_end 88(-4)-88</td>
<td>equidist_end 88(-4)-68</td>
</tr>
<tr>
<td>5</td>
<td>new2</td>
<td>equidist_end 5(1)50</td>
<td>= coordinate definition</td>
</tr>
</tbody>
</table>

Example 11.2 Coordinate transformations by a transformation file

11.3 Value Lists

For variables, coordinates and experiment targets value lists are supplied by the value-item in user-defined files. Value lists describe a sequence of values together with an order. The number of described values has to be greater than 1. Value lists may be restricted to strictly monotonic sequences. They follow the syntax rules in Tab. 11.6.

<table>
<thead>
<tr>
<th>Value-list type</th>
<th>Syntax</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>explicit list</td>
<td>list &lt;real_val1&gt;,..., &lt;real_valn&gt;</td>
<td>explicit list of values same syntax rules as for one record of a file with a value list (see below)</td>
</tr>
<tr>
<td>by reference file</td>
<td>file {&lt;directory&gt;/}&lt;file_name&gt;</td>
<td>file {&lt;directory&gt;/}&lt;file_name&gt; contains the explicit value list</td>
</tr>
<tr>
<td>implicit equidistant with end-element</td>
<td>equidist_end &lt;real_val1&gt; (&lt;real_val2&gt;) &lt;real_val3&gt;</td>
<td>description of an equidistant list of values with begin value &lt;real_val1&gt;, increment &lt;real_val2&gt;, end value &lt;real_val3&gt;, &lt;real_val1&gt; ≠ &lt;real_val3&gt;, &lt;real_val2&gt; ≠ 0.</td>
</tr>
<tr>
<td>implicit equidistant with number of values</td>
<td>equidist_nmb &lt;real_val1&gt; (&lt;real_val2&gt;) &lt;int_val&gt;</td>
<td>description of an equidistant list of values with begin value &lt;real_val1&gt;, increment &lt;real_val2&gt;, number of values &lt;real_val2&gt; ≠ 0, &lt;int_val&gt; &gt; 0</td>
</tr>
</tbody>
</table>

Tab. 11.6 Syntax rules for value lists
Syntax rules for a file {<directory>/}<file_name> with a list of values

- Has to be an ASCII file
- May be a multi-record file
- Max. record length is 1000 characters
- Values 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

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>list 3, 5, 7, 9, 11</td>
<td>describes the five values 3, 5, 7, 9, and 11</td>
</tr>
<tr>
<td>2.</td>
<td>equidist_end 3 (2) 11</td>
<td>is equivalent to 1.</td>
</tr>
<tr>
<td>3.</td>
<td>equidist_end 3 (2) 11.9</td>
<td>is equivalent to 1.</td>
</tr>
<tr>
<td>4.</td>
<td>equidist_nmb 3 (2) 5</td>
<td>is equivalent to 1.</td>
</tr>
<tr>
<td>5.</td>
<td>file my_values.dat</td>
<td>is equivalent to 1. with my_values.dat = 3, , 5, 7, 9, 11</td>
</tr>
<tr>
<td>6.</td>
<td>equidist_end 11 (-2) 3</td>
<td>differs from 1. – 5.: values are identical, ordering sequence differs</td>
</tr>
</tbody>
</table>

**Example 11.3**  **Examples of value lists**
12 Model and Experiment Post-Processor Output Data Structures

This chapter summarizes information on available data structures for model and experiment post-processor output. SimEnv supports several output formats from the experiment and the post-processor. NetCDF is a self-describing data format and can be used for model and post-processor output. Another format specifications for both outputs is IEEE compliant binary format and ASCII for post-processor output. This chapter describes all the used data structures.

Dependent on the specification of the supported experiment post-processor output formats in <model>.cfg model output can be stored in NetCDF format and post-processor output in NetCDF, IEEE or ASCII format. During experiment performance model output is written either to single output files <model>.out<run_char>.[ nc | ascii ] per experiment single run or to a common output file <model>.outall.[ nc | iee | ascii ] 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. <run_char> is a six-digit placeholder for the corresponding single run number.

During experiment post-processing output and structure of results is written to <model>.res<res_char>.[ nc | iee | ascii ] and <model>.res<res_char>.[ iee | ascii ]. <res_char> is a two-digit placeholder for the number of the result file. It ranges from 01 to 99.

For IEEE and ASCII model output and experiment post-processor output formats, multi-dimensional data is organized in the Fortran storage model (see Section 15.7 - Glossary). Use the SimEnv service command simenv.dmp for browsing model and result output files. See Tab. 10.3 for more information.

12.1 NetCDF Model and Experiment Post-Processor Output

The intention for supplying NetCDF format for model and experiment post-processor output is to provide the possibility to generate self-describing, platform-independent data files with metadata that can be interpreted by subsequent visualization techniques. The conventions applied for SimEnv represent a compromise between existing standards and the metadata requirements for a flexible and expressive visualization that is adapted to the requirements of the specific data sets of concern. SimEnv follows the NetCDF Climate and Forecast (NetCDF CF) metadata convention 1.0-beta4. Currently, SimEnv supports only up to 4-dimensional NetCDF output during experiment and post-processor performance.

In principle, any NetCDF file can be viewed by the NetCDF service program

```
ncdump <NetCDF_file>
```

Model output data types as declared in the model output description file <model>.mdf are transferred into NetCDF data types automatically (check the Table below). By default, post-processor output data is of type float.

<table>
<thead>
<tr>
<th>SimEnv data type (see Tab. 5.4)</th>
<th>NetCDF data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>byte</td>
<td>NF_BYTE</td>
</tr>
<tr>
<td>short</td>
<td>NF_SHORT</td>
</tr>
<tr>
<td>int</td>
<td>NF_INT</td>
</tr>
<tr>
<td>float</td>
<td>NF_FLOAT</td>
</tr>
<tr>
<td>double</td>
<td>NF_DOUBLE</td>
</tr>
</tbody>
</table>

**Tab. 12.1** NetCDF data types
12.1.1 Global Attributes

The global attributes used in SimEnv from the CF standard are :institution and :convention. In addition, the following global attributes are defined for model and post-processor output:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>:creation_time</td>
<td>/YYYY-MM-DD HH:MM:SS</td>
<td>char</td>
</tr>
<tr>
<td>:model_name</td>
<td>&lt;model&gt;</td>
<td>char</td>
</tr>
<tr>
<td>:model_description</td>
<td>model output description according to &lt;model&gt;.mdf</td>
<td>char</td>
</tr>
<tr>
<td>:model_description_file</td>
<td>(&lt;directory&gt;/)&lt;model&gt;.mdf</td>
<td>char</td>
</tr>
<tr>
<td>:experiment_type</td>
<td>[ behaviour</td>
<td>monte carlo</td>
</tr>
<tr>
<td>:experiment_description</td>
<td>experiment description according to &lt;model&gt;.edf</td>
<td>char</td>
</tr>
<tr>
<td>:experiment_description_file</td>
<td>(&lt;directory&gt;/)&lt;model&gt;.edf</td>
<td>char</td>
</tr>
<tr>
<td>:number_of_runs</td>
<td>&lt;number of runs&gt;</td>
<td>int</td>
</tr>
</tbody>
</table>

Tab. 12.2 Additional global NetCDF attributes

12.1.2 Variable Labelling and Variable Attributes

For coordinate variables, two cases of labelling are distinguished:
- If for a given predefined variable, target, model output variable or post-processor result one of its coordinates spans the entire range of its general dimension, the already existing coordinate definition is used.
- Otherwise, this concerned coordinate is re-defined using the notation
  <variable_name>_dim_<coordinate_name>.

The following variable attributes are used according to the CF 1.0-beta4 standard:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;variable_name&gt;:standard_name</td>
<td>[ &lt;coordinate_name&gt;</td>
<td>&lt;predef_coordinate_name&gt;</td>
</tr>
<tr>
<td>&lt;variable_name&gt;:long_name</td>
<td>[ &lt;coordinate_description&gt;</td>
<td>&lt;predef_coordinate_description&gt;</td>
</tr>
<tr>
<td>&lt;variable_name&gt;:unit</td>
<td>[ &lt;coordinate_unit&gt;</td>
<td>&lt;predef_coordinate_unit&gt;</td>
</tr>
<tr>
<td>&lt;variable_name&gt;:missing_value</td>
<td>&lt;variable type-depending missing value&gt;</td>
<td>type-dep.</td>
</tr>
<tr>
<td>&lt;variable_name&gt;:axis</td>
<td>[ X</td>
<td>Y</td>
</tr>
<tr>
<td>&lt;variable_name&gt;:coordinates</td>
<td>&lt;par1_lon&gt; &lt;par1_lat&gt;</td>
<td>char</td>
</tr>
<tr>
<td>&lt;variable_name&gt;:_Fillvalue</td>
<td>&lt;variable type-depending fill value&gt;</td>
<td>type-dep.</td>
</tr>
</tbody>
</table>

Tab. 12.3 Variable NetCDF attributes
• For experiment post-processor output, the :standard_name attribute simply counts the number of applied operations because the result name of an arbitrary operation is not known in general. For that reason, the :long_name attribute would re-sample the :standard_name attribute and it is used instead to provide the complete description of the applied operator sequence without defining an additional attribute.

If macros are included, these are resolved and elementary operations are included only.

• For the :axis attribute of a coordinate variable exist defaults.
  For each post-processor result, the first coordinate is assumed to be the „X-axis“, the second and third coordinate are assumed to represent the „Y-“ and „Z-axis“, and the fourth dimension is time T.
  For model results, these attribute values are assigned to coordinate variables describing geographical longitude, geographical latitude, level or height and time. In case other coordinate names are used, these are simply also used for the axis attribute.

• The :unit attribute is actually estimated for model output only depending on the description of the corresponding sub-keywords for the keyword ‘variable’ in the <model>.mdf file. For post-processing output, it is only used as a placeholder and not calculated from the applied operator sequence so far.

• The :coordinates attribute serves to define coordinates depending on other ones and so to allow coordinate transformations. Actually, this attribute is not used.

• Actually, the :_Fillvalue attribute is not applied to coordinate variables. It is identically to the :missing_value attribute but open for other definitions.

For visualization requirements, the following additional variable attributes have been defined for SimEnv:

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;variable_name&gt;:monotony</code> (coordinate variables only)</td>
<td>[ increasing</td>
<td>decreasing</td>
</tr>
<tr>
<td><code>&lt;variable_name&gt;:coo_type</code></td>
<td>[ 1</td>
<td>2 ]</td>
</tr>
<tr>
<td><code>&lt;variable_name&gt;:data_range</code></td>
<td><code>&lt;min&gt; </code>&lt;max&gt;`</td>
<td>char</td>
</tr>
<tr>
<td><code>&lt;variable_name&gt;:index_range_&lt;coordinate&gt;</code> (coordinate variables only)</td>
<td><code>&lt;min_index&gt; </code>&lt;max_index&gt;</td>
<td>int</td>
</tr>
<tr>
<td><code>&lt;variable_name&gt;:simenv_data_kind</code></td>
<td>[ predefined model output variable</td>
<td>model target</td>
</tr>
<tr>
<td><code>&lt;variable_name&gt;:var_representation</code></td>
<td>[ positions</td>
<td>connections</td>
</tr>
<tr>
<td><code>&lt;variable_name&gt;:grid_shift</code></td>
<td><code>&lt;shift_x&gt; </code>&lt;shift_y&gt;</td>
<td>real, dimension(2)</td>
</tr>
<tr>
<td><code>&lt;variable_name&gt;:north_pole</code></td>
<td><code>&lt;lon_pole&gt; </code>&lt;lat_pole&gt;</td>
<td>real, dimension(2)</td>
</tr>
</tbody>
</table>

Tab. 12.4 Variable NetCDF attributes for visualization

• The :monotony attribute is applied to coordinate variables only and estimated from the coordinate values as defined in the <model>.mdf file. During post-processing additional coordinates can be generated for which no monotony may be estimated. In such cases, the attribute is set to "none".

• The :coo_type attribute describes the grid representation of a given coordinate. A value of 1 indicates that all coordinate values are provided explicitly (suitable, e.g., for irregular grids). A value of 2 indicates a regular grid and a coordinate representation by its start value, increment and end value.

• The :data_range attribute provides the real range that is covered by the related variable in the recent NetCDF file.

• The :index_range attribute is used only in case a predefined output variable, target, model output variable or post-processing result covers not the complete range of a dimension as defined for a coordinate variable. It describes that sub-space for which the concerned target, variable or result is defined.

• The :var_representation attribute is introduced to specify what operations are allowed on the data.

• The :grid_shift attribute is actually still a placeholder for variables that are not defined in the centre of a grid box when quasi-regular grids are used.

• The :north_pole attribute can be used if rotated grids are applied.
12.2 IEEE Compliant Binary Model Output

IEEE compliant binary model output is written in records of fixed length to `<model>.out<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 targets as specified in `<model>.edf`
  Sequence as in `<model>.edf`
- Built-in (pre-defined) model output variables
  Sequence as in Tab. 10.8
- Model output variables
  Sequence as in `<model>.mdf`

Storage demand for each model output variable / target is according to its dimensionality, extents and data type. Storage demand in bytes for each model output variable / target is readjusted to the smallest number of bytes divisible by 8, where the data can be stored. Multi-dimensional data fields are organized in the Fortran storage model (see Section 15.7 - Glossary).

Data is stored in records with a fixed record length of minimum( 512000 Bytes , readjusted storage demand in Bytes ).
In `<model>.outall.ieee` each single run starts with a new record. Sequence of single runs corresponds with sequence of the single run numbers `<run>`. Consequently, data from default single run 0 is stored in the first and potentially the following records.

<table>
<thead>
<tr>
<th>Target / model variable</th>
<th>Extents</th>
<th>Data type</th>
<th>Storage demand [Byte]</th>
<th>Storage demand adjusted [Byte]</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>1</td>
<td>float</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>p2</td>
<td>1</td>
<td>float</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>sim_time</td>
<td>1</td>
<td>float</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>atmo</td>
<td>45 x 90 x 4 x 20</td>
<td>float</td>
<td>1.296.000</td>
<td>1.296.000</td>
</tr>
<tr>
<td>bios</td>
<td>36 x 90 x 20</td>
<td>float</td>
<td>259.200</td>
<td>259.200</td>
</tr>
<tr>
<td>atmo_g</td>
<td>20</td>
<td>int</td>
<td>80</td>
<td>80</td>
</tr>
<tr>
<td>bios_g</td>
<td>1</td>
<td>int</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1.555.312</td>
</tr>
</tbody>
</table>

One single run needs 1.555.312 : 512.000 = 3+1 records with a fixed length of 512.000 Bytes. Remaining bytes in the last record are undefined.

**Example 12.1**  
*IEEE compliant model output data structure*
12.3 IEEE Compliant Binary and ASCII Experiment Post-Processor Output

For IEEE and ASCII experiment post-processor output result information is stored in two files:

- `<model>.res<res_char>.[ ieee | ascii ]` holds the result dynamics
- `<model>.inf<res_char>.[ ieee | ascii ]` holds structure and coordinate information

The IEEE post-processor output files `<model>.res<res_char>.ieee` and `<model>.inf<res_char>.ieee` are unformatted binary files with IEEE float / int number representation, while for the ASCII post-processor version `<model>.res<res_char>.ascii` and `<model>.inf<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<res_char>.[ ieee | ascii ]` for each result:**

result number 01:

- record no. 1: max. 512 chars result expression string
- record no. 2: max. 128 chars result description string
- record no. 3: max. 32 chars result unit string (or 1 space if unit is undefined)
- record no. 4: 10 int dim ext(1) ... ext(dim) 0 ... 0
- record no. 5: max. 20 chars coordinate name of dimension 1
- record no. 6: 10 float coordinate values of dimension 1 in records of 10 values
  (last record may have less values)

...

result number xxx:

- record no. xxx: max. 20 chars coordinate name of dimension dim
- record no. xxx+1: 10 float coordinate values of dimension dim in records of 10 values
  (last record may have less values)

result number 02:

...

**Record structure of `<model>.res<res_char>.[ ieee | ascii ]` for each result:**

result number 01:

- record no. 1: 10 float in records of 10 values (last record may have less values):
  result_value(1) ... result_value(length_result)
  with `length_result = \prod_{i=1}^{dim} ext(i)` for dim > 0
  = 1 else

result number 02:

...

The vector `result_value` is stored in the Fortran storage model (see Section 15.7 - Glossary). The nodata element for undefined result values is set to 3.4E38.

The Fortran code in Example 15.11 reads experiment post-processing ASCII output files `<model>.res<res_char>.ascii` and `<model>.inf<res_char>.ascii` in their general structure. In the examples-directory `$SE_HOME/..examples` of SimEnv it is accompanied by the corresponding version for IEEE result output.
13 SimEnv Prospects

SimEnv development and improvement is user-driven. Here one can find a list of the main development pathways in future.

General
- Graphical user interface
- Portability to Windows-based systems
- Unique number representations for binary output of distributed models (big endians vs. small endians)

Model interface
- Interface for Java models
- simenv_slice_py for Python models

Experiment preparation
- Experiment type qualitative global sensitivity analysis
- Experiment type uncertainty analysis with variance decomposition
- Experiment type stochastic analysis
- Monte Carlo analysis: stopping rules and sampling of correlated targets

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 and global sensitivity analyses 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

Experiment evaluation
- Advanced techniques for graphical representation of experiment post-processor output, especially for multi-run operators
14 References

http://ic.net/~jnbohr/java/CdfDemoMain.html

http://www.jrc.eu.int/uasa/primer-sa.asp

No. 47, Potsdam Institute for Climate Impact Research, Potsdam
http://www.pik-potsdam.de/reports/pr-47/pr47.pdf

Output. Proceedings of the 4th International Conference on Sensitivity Analysis of Model Output (SAMO
http://library.lanl.gov/ccw/samo2004/

Methods. Sandia National Laboratories, Albuquerque, U.S.A.

In: Saltelli et al. (2000)

Computer Models. Risk Anal. 8(1), 71-90


sessing Scientific Models. J. Wiley & Sons, Chichester

Schulzweida, U. (2004): Climate Data Operators. Max-Planck-Institute for Meteorology, Hamburg and
http://www.mpimet.mpg.de/~cdo

objects. Max-Planck-Institute for Meteorology, Hamburg and
http://www.mad.zmaw.de/Pingo/pingohome.html

PIK-Report No. 13, Potsdam Institute for Climate Impact Research, Potsdam
http://www.pik-potsdam.de/pik_web/publications/pik_reports/sum_pr13.htm


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 implemented under Unix and Linux. The Unix version is available at PIK from /usr/local/simenv. The Linux version is directly available from the SimEnv developers. For both versions, only the latest version is supported and bug fixes are installed on demand. Tab. 15.1 lists the directory structure of SimEnv.

<table>
<thead>
<tr>
<th>Sub-directory of /usr/local/bin</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>latest version of SimEnv</td>
</tr>
<tr>
<td>doc</td>
<td>documentation for the latest version</td>
</tr>
<tr>
<td>examples</td>
<td>example files for the latest version</td>
</tr>
<tr>
<td>version_archive</td>
<td>version archive of SimEnv, structured for each version in the same manner as for the latest version</td>
</tr>
</tbody>
</table>

Tab. 15.1 SimEnv installation under Unix

15.1.1 System Requirements

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>hardware</td>
<td>RS6000 and compatibles Intel-based systems and compatibles</td>
</tr>
<tr>
<td>operating system</td>
<td>AIX Version 4.3 or higher SUSE Version 9.0 or higher <a href="http://www.suse.com">http://www.suse.com</a></td>
</tr>
<tr>
<td>shell</td>
<td>Bourne shell sh</td>
</tr>
<tr>
<td>Fortran compiler (only for compiling and linking inter-faced Fortran models and user-defined operators)</td>
<td>xlf IBM Fortran compiler ifort Intel Fortran compiler</td>
</tr>
<tr>
<td>C/C++ compiler (only for compiling and linking inter-faced models and user-defined operators written in C/C++)</td>
<td>xlc IBM C/C++ compiler gcc GNU C/C++ compiler</td>
</tr>
<tr>
<td>Python</td>
<td>Version 2.3 or higher <a href="http://www.python.org">http://www.python.org</a></td>
</tr>
<tr>
<td>OpenDX (Linux-based visualization server)</td>
<td>Version 4.3.2 or higher <a href="http://www.opendx.org">http://www.opendx.org</a></td>
</tr>
<tr>
<td>NetCDF-CF</td>
<td>Version 1.04 or higher <a href="http://www.cgd.ucar.edu/cms/eaton/cf-metadata">http://www.cgd.ucar.edu/cms/eaton/cf-metadata</a></td>
</tr>
</tbody>
</table>

Tab. 15.2 System requirements
## 15.1.2 Technical Limitations

<table>
<thead>
<tr>
<th>Entity</th>
<th>Limitation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User-defined files entities (check also Section 11.1)</strong></td>
<td></td>
</tr>
<tr>
<td>max. length of a record in a user-defined file</td>
<td>[characters] 160</td>
</tr>
<tr>
<td>max. length of all general descriptions descr</td>
<td>[characters] 512</td>
</tr>
<tr>
<td>max. length of a local description descr</td>
<td>[characters] 128</td>
</tr>
<tr>
<td>max. length of a unit</td>
<td>[characters] 32</td>
</tr>
<tr>
<td>max. length of a name</td>
<td>[characters] 20</td>
</tr>
<tr>
<td>max. sum from user-defined and composed operators in &lt;model&gt;.odf</td>
<td>45</td>
</tr>
<tr>
<td>max. length of all define strings for a macro or a composed operator</td>
<td>[characters] 512</td>
</tr>
<tr>
<td>max. length of a record of a referred data file</td>
<td>[characters] 1 000</td>
</tr>
</tbody>
</table>

| **Model interface and experiment preparation entities**                |            |
| max. dimensionality of a model output variable                       | 9          |
| max. dimensionality of a model output variable for Python models     | 4          |
| max. dimensionality of a model output variable for GAMS models       | 4          |
| max. dimensionality of a model output variable stored in NetCDF format | 4          |
| max. number of model output variables                                | 50         |
| max. number of coordinates                                          | 30         |
| max. number of experiment targets                                    | 50         |
| max. number of slice definitions during interfacing a model          | 30         |
| max. number of single model runs in an experiment                    | 999 999    |
| max. number of coordinate values and target adjustment values        | 200 000    |

| **Experiment post-processing entities (per result)**                 |            |
| max. length of the optional result description string                | [characters] 128 |
| max. length of the optional result unit string                       | [characters] 32 |
| max. number of arguments of an operator                              | 9          |
| max. dimensionality of a result                                      | 9          |
| max. dimensionality of a result stored in NetCDF format              | 4          |
| max. length of a complete result string (with description and unit)  | [characters] 512 |
| max. number of all operands and operators of a result                | 200        |
| max. length of a string for a constant                               | [characters] 20 |
| max. number of constants                                             | 30         |
| max. number of allocatable main memory segments                      | 10         |
| max. allocatable main memory                                         | [MBytes] 240 |
| max. number of post-processor output files                           | 99         |

*Tab. 15.3*  
Current SimEnv technical limitations

### 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
  - `$SE_HOME/libsimenv.a`
  - `/usr/local/lib/libnetcdf.a`
- User-defined operators to be used in experiment post-processing have to be linked with the following library to interface them to the simulation environment
  - `$SE_HOME/libsimenv.a`

For running interfaced models outside SimEnv check Section 5.10.
15.1.4 Example Models and User Files

For the following models corresponding files of Tab. 10.6 of can be copied from the corresponding examples-directory $SE_HOME/..../examples to the user's current workspace by running the SimEnv service command simenv.cpy <model> from this workspace:

<table>
<thead>
<tr>
<th>model</th>
<th>Language / source code</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>world_f</td>
<td>Fortran world_f.f</td>
<td>global atmosphere - biosphere model at resolution of ( lat x lon x level x time ) = ( 45 x 90 x 4 x 20 )</td>
</tr>
<tr>
<td>world_c</td>
<td>C world_c.c</td>
<td></td>
</tr>
<tr>
<td>world_cpp</td>
<td>C++ world_cpp.cpp</td>
<td></td>
</tr>
<tr>
<td>world_py</td>
<td>Python world_py.py</td>
<td></td>
</tr>
<tr>
<td>world_sh</td>
<td>Shell script level world_sh.f world_shput.f</td>
<td></td>
</tr>
<tr>
<td>world_f_auto</td>
<td>Fortran world_f_auto.f</td>
<td></td>
</tr>
<tr>
<td>world_sh_auto</td>
<td>Shell script level world_sh.f world_shput.f</td>
<td></td>
</tr>
<tr>
<td>world_f_1x1</td>
<td>Fortran world_f_1x1.f</td>
<td>global atmosphere - biosphere model at a resolution of ( lat x lon x level x time ) = ( 180 x 360 x 16 x 20 )</td>
</tr>
<tr>
<td>world_f_05x05</td>
<td>Fortran world_f_05x05.f</td>
<td>global atmosphere - biosphere model at a resolution of ( lat x lon x level x time ) = ( 360 x 720 x 16 x 20 )</td>
</tr>
<tr>
<td>gridcell_f</td>
<td>Fortran gridcell_f.f</td>
<td>global atmosphere - biosphere model for one lat-lon grid cell at a resolution of ( level x time ) = ( 4 x 20 )</td>
</tr>
<tr>
<td>gams_model</td>
<td>GAMS gams_model.gms</td>
<td>GAMS example model</td>
</tr>
</tbody>
</table>

Tab. 15.4 Implemented example models for the current version

For the generic model "world" check Example 1.1

Additionally, the following files are available from the example directory $SE_HOME/..../examples:

<table>
<thead>
<tr>
<th>File</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;model&gt;[f</td>
<td>c</td>
</tr>
<tr>
<td>&lt;model&gt;</td>
<td>model executable compiled and linked from &lt;model&gt;[f</td>
</tr>
<tr>
<td>world.edf[a</td>
<td>b</td>
</tr>
<tr>
<td>world.post_[c</td>
<td>e</td>
</tr>
<tr>
<td>world.dat[d</td>
<td>e</td>
</tr>
<tr>
<td>usr_opr_&lt;opr&gt;.f</td>
<td>source code for user-defined operator &lt;opr&gt;</td>
</tr>
<tr>
<td>&lt;opr&gt;.opr</td>
<td>executable for user-defined operator &lt;opr&gt;</td>
</tr>
<tr>
<td>usr_opr_&lt;opr&gt;.f</td>
<td>source code file for user-defined post-processing operator &lt;opr&gt;</td>
</tr>
</tbody>
</table>
### 15.1.5 Example User-Defined Operators

The following user-defined operators are available from the example directory $SE_HOME/..examples as source code and executables <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.

<table>
<thead>
<tr>
<th>Operator name &lt;opr&gt;</th>
<th>Operator arguments</th>
<th>Explanation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>char_test</td>
<td>char_arg1,char_arg2, arg</td>
<td>character test check usr_opr_char_test.f</td>
<td>char_test('arg11', 'arg22', bios)</td>
</tr>
<tr>
<td>corr_coeff</td>
<td>arg1,arg2</td>
<td>correlation coefficient R</td>
<td>corr_coeff(bios, -bios) = -1.</td>
</tr>
<tr>
<td>div</td>
<td>arg1,arg2</td>
<td>division as an example how the corresponding built in basic operator works</td>
<td>div(-2,-4) = 0.5</td>
</tr>
<tr>
<td>matmul_ [f</td>
<td>c ]</td>
<td>arg1,arg2</td>
<td>matrix multiplication of 2-dimensional operands</td>
</tr>
<tr>
<td>simple_div</td>
<td>arg1,arg2</td>
<td>division without consideration of overflow, underflow, and division by 0.</td>
<td>simple_div(-2,-4) = 0.5</td>
</tr>
</tbody>
</table>

**Tab. 15.5** Implemented model and operator related user files for the current version

For <opr> see Tab. 15.6 below

**Tab. 15.6** Available user-defined operators
15.2 Examples for Model Interfaces

15.2.1 Example Implementation of the Generic Model world

According to Example 1.1 on page 4 dynamics of the model world depend on four model parameters p1, p2, p3, and p4:

<table>
<thead>
<tr>
<th>Model target</th>
<th>Target default value</th>
<th>Internal model Parameter name</th>
<th>Target unit</th>
<th>Target meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>1.</td>
<td>phi_lat</td>
<td>π/12</td>
<td>latitudinal phase shift</td>
</tr>
<tr>
<td>p2</td>
<td>2.</td>
<td>omega_lat</td>
<td>2*π</td>
<td>latitudinal frequency</td>
</tr>
<tr>
<td>p3</td>
<td>3.</td>
<td>phi_lon</td>
<td>π/12</td>
<td>longitudinal phase shift</td>
</tr>
<tr>
<td>p4</td>
<td>4.</td>
<td>omega_lon</td>
<td>2*π</td>
<td>longitudinal frequency</td>
</tr>
</tbody>
</table>

Tab. 15.7 Targets of the generic model world

Mapping between model targets and internal model parameters is performed by the model coupling interface functions simenv_get_*

For reasons of simplification these targets (parameters) influence state variables atmo and bios by the product of two trigonometric terms value_lat and value_lon in the following manner:

\[
\text{value}_{\text{lat}}(\text{lat}) = \sin\left(2\pi \omega_{\text{lat}} f(\text{lat}) + \phi_{\text{lat}}\pi/12\right)
\]

\[
\text{value}_{\text{lon}}(\text{lon}) = \sin\left(2\pi \omega_{\text{lon}} f(\text{lon}) + \phi_{\text{lon}}\pi/12\right)
\]

The function \( f( . ) \) norms value_lat and value_lon by lat and/or lon in a way, that it holds

\[
\text{value}_{[\text{lat}|\text{lon}]}(1) = \sin\left(+n\omega_{[\text{lat}|\text{lon}]} + \phi_{[\text{lat}|\text{lon}]}/12\right)
\]

\[
\text{value}_{[\text{lat}|\text{lon}]}(\text{last}/2) = \sin\left(\pm0\omega_{[\text{lat}|\text{lon}]} + \phi_{[\text{lat}|\text{lon}]}/12\right)
\]

\[
\text{value}_{[\text{lat}|\text{lon}]}(\text{last}) = \sin\left(-n\omega_{[\text{lat}|\text{lon}]} + \phi_{[\text{lat}|\text{lon}]}/12\right)
\]

Finally,

\[
\text{atmo}(\text{lat},\text{lon},\text{level},\text{time}) = \text{value}_{\text{lat}}(\text{lat}) \times \text{value}_{\text{lon}}(\text{lon}) \times (100\times\text{time}+\text{level}-1)
\]

\[
\text{bios}(\text{lat},\text{lon},\text{time}) = \text{value}_{\text{lat}}(\text{lat}) \times \text{value}_{\text{lon}}(\text{lon}) \times 100\times\text{time}
\]

and

\[
\text{atmo}_g(\text{time}) = \text{avg}_l('001',\text{abs}(\text{atmo}(\text{lat},\text{lon},1,\text{time})))
\]

\[
\text{bios}_g = \text{avg}(\text{abs}(\text{bios}(\text{lat},\text{lon},\text{time})))
\]

Means \( \text{avg} \) and \( \text{avg}_l \) are calculated in a box with the extent \( \Delta \text{lat} \times \Delta \text{lon} = 10^\circ \times 10^\circ \) and \( (\text{lat},\text{lon}) = (0^\circ,0^\circ) \) 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**.

```fortran
program world_f
  c declare SimEnv interface functions (compile with -I$SE_HOME)
  c simenv_sts, simenv_run_int and simenv_run_char are also declared there
   include ‘simenv_mod_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.

  simenv_sts = simenv_ini_f()
  c check return code for the model interface functions at least here
   if(simenv_sts.ne.0) call exit_(1)
  c only if necessary:
   simenv_sts = simenv_get_run(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
    ... c 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
  ... c bios_g
   simenv_sts = simenv_put_f(‘bios_g’,bios_g)
  simenv_sts = simenv_end_f()
end
```

Example file: **world_f.f**

**Example 15.1**  
Model interface for Fortran models - model **world_f.f**
15.2.3 Fortran Model with Semi-Automated Model Interface

With respect to Example 5.1 the following Fortran code `world_f_auto.f` could be used to describe the model interfaced semi-automatically to SimEnv. SimEnv modifications are marked in bold.

```fortran
program world_f_auto
  c declare SimEnv interface functions (compile with -I$SE_HOME)
  c simenv_sts, simenv_run_int and simenv_run_char are also declared there
  include ‘simenv_mod_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 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.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
/* declare SimEnv interface functions (compile with `-I$SE_HOME`
 simenv_sts, simenv_run_int and simenv_run_char are also declared there */
#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,levell,idecade1,idim;
    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;
    /* 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;
            levell=level+1;
            simenv_sts = simenv_slice_c("atmo",&idim,&levell,&levell);
            idim=4;
            idecade1=idecade+1;
            simenv_sts = simenv_slice_c("atmo",&idim,&idecade1,&idecade1);
            simenv_sts = simenv_put_c("atmo",(char *) &atmo);
        }
    }
}
```

Example file: world_c.c

Example 15.3  Model interface for C models – model world_c.c

```c
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
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**.

```c++
#include <stdio.h>
#include <stdlib.h>
/* declare SimEnv interface functions (compile with -I$SE_HOME) 
   simenv_sts, simenv_run_int and simenv_run_char are also declared there */
#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,levell,idecadel,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;
           levell=levell+1;
           simenv_sts = simenv_slice_c("atmo",&idim,&level,&level);
           idim=4;
           idecadel=idecadel+1;
           simenv_sts = simenv_slice_c("atmo",&idim,&idecadel,&idecadel);
           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 */
  { for (idecade=0; idecade<=19; idecade++)
      {...
       idim=3;
       idecadel=idecadel+1;
       simenv_sts = simenv_slice_c("bios",&idim,&idecadel,&idecadel);
       simenv_sts = simenv_put_c("bios",(char *) &bios);
     } /* compute dynamics of bios_g */
    ...
  }
};
```
main(void)
{
  float p1 = 1.;
  float p2 = 2.;
  float p3 = 3.;
  float p4 = 4.;

  simenv_sts = simenv_init_c();
  /* check return code of model interface functions at least here */
  if(simenv_sts != 0) return 1;
  /* 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;
}
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**.

```python
#!/usr/local/bin/python
import string
import os
from simenv import *
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()
# 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 15.5** Model interface for Python models – model `world_py.py`
15.2.7 Model Interface at Shell Script Level

Assume any experiment. Assume model executable world_sh to take target 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:

```sh
#!/bin/sh

p1=1.
p2=2.
p3=3.
p4=4.

# perform always and as the first $SE_HOME/simenv_*_sh dot script
# altern. perform . $SE_WS/<model>_sh.inc for semi-autom. model interface
. $SE_HOME/simenv_ini_sh

# get current run number simenv_run_char and simenv_run_int
. $SE_HOME/simenv_get_run_sh

# get adjustments for p1 ... p4
target_name='p1'
target_def_val=$p1
. $SE_HOME/simenv_get_sh

target_name='p2'
target_def_val=$p2
. $SE_HOME/simenv_get_sh

target_name='p3'
target_def_val=$p3
. $SE_HOME/simenv_get_sh

target_name='p4'
target_def_val=$p4
. $SE_HOME/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/simenv_*_sh dot script
. $SE_HOME/simenv_end_sh
```

Example file: world_sh.run

Example 15.6 Model interface at shell script level – model shell script world_sh.run
15.2.8 Semi-Automated Model Interface at Shell Script Level

Assume any experiment. Assume model executable world_sh to take target values p1 to p4 as arguments from the command line.

The shell script `world_sh_auto.run` with an semi-automated interface at shell script level to run the model world_sh and to transform model output to SimEnv could look like:

```bash
#!/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/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/simenv_*_sh dot script
# . $SE_HOME/simenv_end_sh
```

*Example file: world_sh_auto.run*

**Example 15.7**  Semi-automated model interface at shell script level – model shell script world_sh_auto.run
15.2.9 GAMS Model

SimEnv comes with an interfaced GAMS model `gams_model.gms` and all associated files that fully correspond with the GAMS example model at http://www.gams.com/docs/gams/Tutorial.pdf. Modifications for SimEnv are marked in **bold**.

```gams
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 parameter (here: dem_ny and dem_ch) as SimEnv experiment targets they have to be declared as GAMS model parameters default values from above.
* - Then insert $include <model>_simenv_get.inc simenv_get.inc is generated automatically based on <model>.edf
* - and assign adjusted targets to model output 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 (see Example 5.5):
  * $call "gams ../sub_m1.gms ll= lo=2 1f=gams_model.nlog dp=0";
  * $call "gams ../sub_m2.gms ll= lo=2 1f=gams_model.nlog dp=0";

Example file: gams_model.gms

Example 15.8 Model interface for GAMS models – model gams_model.gms
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`:

```fortran
integer*4 function simenv_check_user_def_operator()
c declare SimEnv interface functions (compile with -I$SE_HOME)
  include 'simenv_opr_f.inc'
c declare fields to hold extents and coordinates
dimension iext1(9),iext2(9)
dimension ico_nr1(9),ico_nr2(9)
dimension ico_beg_pos1(9),ico_beg_pos2(9)
character*20 co_name1(9),co_name2(9)
c get dimensionality idimens, extents iext,
c formal coordinate number ico_nr and

c formal coordinate begin position ico_beg_pos
idimens1=simenv_get_dim_arg_f(1,iext1)
idimens2=simenv_get_dim_arg_f(2,iext2)
iok=simenv_get_co_arg_f(1,ico_nr1,ico_beg_pos1,co_name1)
iok=simenv_get_co_arg_f(2,ico_nr2,ico_beg_pos2,co_name2)
c get check modus for coordinates
ichk_modus=simenv_get_co_chk_modus_f()
  if(idimens1.ne.2.or.idimens2.ne.2) then
    c wrong dimensionalities
    ierror=1
  else
    if(iext1(2).ne.iext2(1)) then
      c wrong extents
      ierror=2
    else
      if(ico_nr1(2).eq.ico_nr2(1)) then
        c coordinates Identical
        if(ico_beg_pos1(2).eq.ico_beg_pos2(1)) then
          iret=31
        else
          iret=33
        endif
     endif
      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
deddo
dendif
dendif

eerror=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
dendif
dendif
dendif

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)
dendif
c  return error code
simenv_check_user_def_operator=ierror
return
dend

integer*4 function simenv_compute_user_def_operator(res)
c  SimEnv operator results are always of type real*4
real*4 res(1)
c declare SimEnv interface functions (compile with -I$SE_HOME)
include 'simenv_opr_f.inc'
c auxiliary variables
integer*4 iext1(9),iext2(9)
real*8 value8

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
  if(simenv_chk_undef_f(fac1)+simenv_chk_undef_f(fac2).eq.0)
thenn
Example 15.9  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*:

```c
#include <strings.h>
#include <stdio.h>
#include "simenv_opr_c.inc"  /* compile with -I$SE_HOME */

int simenv_check_user_def_operator()
{
    int iext1[9],iext2[9];
    int ico_nr1[9],ico_nr2[9],ico_beg_pos1[9],ico_beg_pos2[9];
    char co_name1[180],co_name2[180];
    int idimens1, idimens2;
    int ichk_modus;
    int iret,iretv1,iretv2,j,iok,ierror=0;
    float value1, value2;

    /* get dimensionality idimens, extents iext, formal coordinate number ico_nr and formal coordinate begin position ico_beg_pos */
    idimens1=simenv_get_dim_arg_c(1,iext1);
    idimens2=simenv_get_dim_arg_c(2,iext2);
    iok=simenv_get_co_arg_c(1,ico_nr1,ico_beg_pos1,co_name1);
    iok=simenv_get_co_arg_c(2,ico_nr2,ico_beg_pos2,co_name2);

    ichk_modus=simenv_get_co_chk_modus_c();

    if(idimens1!=2 || idimens2!=2)  /* wrong dimensionalities */
        ierror=1;
    else
        if(iext1[1]!=iext2[0])  /* wrong dimensions */
            ierror=2;
        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(ierr==0)
    { iext1[1]=iext2[1];
      ico_nr1[1]=ico_nr2[1];
      ico_beg_pos1[1]=ico_beg_pos2[1];
      iok=simenv_put_struct_res_c(0,idimens1,iext1,ico_nr1,
                                  ico_beg_pos1);
    }
    return ierror; /* return error code */

/* SimEnv operator results are always of type real*4 */
int simenv_compute_user_def_operator(float *res)
{
    int iext1[9],iext2[9];
    double value8;
    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;
                              value8=value8+fac1*fac2;
                            }
                    }
                m=m+1;
              }
Example 15.10  

Experiment post-processor user-defined operator module – operator matmul_c

```c
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
15.4 Example for an Experiment Post-Processor Result Import Interface

In Example 15.11 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.

```fortran
subroutine read_result_file_ascii(model_name,res_nmb)
character model_name*20,res_nmb*2
real*4, pointer, dimension(:) :: coord_values
real*4, pointer, dimension(:) :: result_values
integer*4 idim, iext(9)
character result_expr*512, result_desc*128, result_unit*32
character coord_name*20
open(unit=1,file=trim(model_name)//'inf'//res_nmb//'.'.ascii',
#      form='formatted',status='old')
open(unit=2,file=trim(model_name)//'res'//res_nmb//'.'.ascii',
#      form='formatted',status='old')
iostat=0
do while (iostat.eq.0)
read(1,'(a512)',iostat=iostat) result_expr
if(iostat.eq.0) then
read(1,'(a128)',iostat=iostat1) result_desc
read(1,'(a32)',iostat=iostat1) result_unit
read(1,'(10i8)',iostat=iostat1) idim,(iext(i),i=1,9)
length_result=1
do i=1,idim
  length_result=length_result*iext(i)
read(1,'(a20)',iostat=iostat1) coord_name
allocate(coord_values(iext(i)))
ibeg=1
do while (ibeg.le.iext(i))
  iend=min0(ibeg+9,iext(i))
  read(1,'(10g12.6)',iostat=iostat1) (coord_values(j),
    ibeg=iend+1                        j=ibeg,iend)
  c           further processing of coordinate values
  c           ...
  deallocate (coord_values)
endo
distribute(coord_values(length_result))
ibeg=1
do while (ibeg.le.length_result)
  iend=min0(ibeg+9,length_result)
  read(2,'(10g12.6)',iostat=iostat) (result_values(j),
    ibeg=iend+1                        j=ibeg,iend)
  c           further processing of result values
  c           ...
  deallocate(result_values)
endo
close(unit=1)
close(unit=2)
return
end
```

Example file: `read_result_file.f` (together with subroutine `read_result_file_ieee`)

Example 15.11  ASCII compliant experiment post-processor result import interface
15.5 List of Experiment Post-Processor Built-In Operators and Operator Arguments

15.5.1 Experiment Post-Processor Built-In Operators (in Thematic Order)

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Elemental operators</strong></td>
<td></td>
<td>Tab. 8.3 on page 70</td>
</tr>
<tr>
<td>arg1 + arg2</td>
<td>addition</td>
<td></td>
</tr>
<tr>
<td>arg1 - arg2</td>
<td>subtraction</td>
<td></td>
</tr>
<tr>
<td>arg1 * arg2</td>
<td>multiplication</td>
<td></td>
</tr>
<tr>
<td>arg1 / arg2</td>
<td>division</td>
<td></td>
</tr>
<tr>
<td>arg1 ** arg2</td>
<td>exponentiation</td>
<td></td>
</tr>
<tr>
<td>+ arg</td>
<td>identity</td>
<td></td>
</tr>
<tr>
<td>- arg</td>
<td>negation</td>
<td></td>
</tr>
<tr>
<td>( arg )</td>
<td>parentheses</td>
<td></td>
</tr>
<tr>
<td><strong>Basic operators</strong></td>
<td></td>
<td>Tab. 8.4 on page 71</td>
</tr>
<tr>
<td>abs(arg)</td>
<td>absolute value</td>
<td></td>
</tr>
<tr>
<td>dim(arg1,arg2)</td>
<td>positive difference</td>
<td></td>
</tr>
<tr>
<td>exp(arg)</td>
<td>exponential function</td>
<td></td>
</tr>
<tr>
<td>int(arg)</td>
<td>truncation value</td>
<td></td>
</tr>
<tr>
<td>log(arg)</td>
<td>natural logarithm</td>
<td></td>
</tr>
<tr>
<td>log10(arg)</td>
<td>decade logarithm</td>
<td></td>
</tr>
<tr>
<td>mod(arg1,arg2)</td>
<td>remainder</td>
<td></td>
</tr>
<tr>
<td>nint(arg)</td>
<td>round value</td>
<td></td>
</tr>
<tr>
<td>sign(arg)</td>
<td>sign of value</td>
<td></td>
</tr>
<tr>
<td>sqrt(arg)</td>
<td>square root</td>
<td></td>
</tr>
<tr>
<td><strong>Trigonometric operators</strong></td>
<td></td>
<td>Tab. 8.4 on page 71</td>
</tr>
<tr>
<td>sin(arg)</td>
<td>sine</td>
<td></td>
</tr>
<tr>
<td>cos(arg)</td>
<td>cosine</td>
<td></td>
</tr>
<tr>
<td>tan(arg)</td>
<td>tangent</td>
<td></td>
</tr>
<tr>
<td>cot(arg)</td>
<td>cotangent</td>
<td></td>
</tr>
<tr>
<td>asin(arg)</td>
<td>arc sine</td>
<td></td>
</tr>
<tr>
<td>acos(arg)</td>
<td>arc cosine</td>
<td></td>
</tr>
<tr>
<td>atan(arg)</td>
<td>arc tangent</td>
<td></td>
</tr>
<tr>
<td>acot(arg)</td>
<td>arc cotangent</td>
<td></td>
</tr>
<tr>
<td>sinh(arg)</td>
<td>hyperbolic sine</td>
<td></td>
</tr>
<tr>
<td>cosh(arg)</td>
<td>hyperbolic cosine</td>
<td></td>
</tr>
<tr>
<td>tanh(arg)</td>
<td>hyperbolic tangent</td>
<td></td>
</tr>
<tr>
<td>coth(arg)</td>
<td>hyperbolic cotangent</td>
<td></td>
</tr>
<tr>
<td><strong>Advanced operators</strong></td>
<td></td>
<td>Tab. 8.8 on page 77</td>
</tr>
<tr>
<td>classify(int_arg1, real_arg2,real_arg3,arg4)</td>
<td>classification of arg4 into int_arg1 classes</td>
<td></td>
</tr>
<tr>
<td>clip(char_arg1,arg2)</td>
<td>clip arg2 according to char_arg1</td>
<td></td>
</tr>
<tr>
<td>cumul(char_arg1,arg2)</td>
<td>cumulates arg2 according to char_arg1</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>See</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>-----</td>
</tr>
<tr>
<td>flip(char_arg1, arg2)</td>
<td>flip arg2 according to char_arg1</td>
<td></td>
</tr>
<tr>
<td>get_data(char_arg1, char_arg2, char_arg3, arg4)</td>
<td>get data from an external file</td>
<td></td>
</tr>
<tr>
<td>get_experiment(char_arg1, char_arg2, char_arg3, arg4)</td>
<td>include an other experiment</td>
<td></td>
</tr>
<tr>
<td>get_table_fct(char_arg1, arg2)</td>
<td>table function with linear interpolation of table char_arg1 for position arg2</td>
<td></td>
</tr>
<tr>
<td>if(char_arg1, arg2, arg3, arg4)</td>
<td>general purpose conditional if-construct</td>
<td></td>
</tr>
<tr>
<td>mask(char_arg1, arg2, arg3)</td>
<td>mask elements of argument arg2</td>
<td></td>
</tr>
<tr>
<td>matmul(arg1, arg2)</td>
<td>matrix multiplication</td>
<td></td>
</tr>
<tr>
<td>move_avg(char_arg1, char_arg2, int_arg3, arg4)</td>
<td>moving average of running length int_arg3 for arg4</td>
<td></td>
</tr>
<tr>
<td>nr_of_runs()</td>
<td>number of single runs of the current experiment</td>
<td></td>
</tr>
<tr>
<td>rank(char_arg1, arg2)</td>
<td>rank of arg2 according to char_arg1</td>
<td></td>
</tr>
<tr>
<td>regrid(char_arg1, arg2)</td>
<td>assign new coordinates to arg2</td>
<td></td>
</tr>
<tr>
<td>run(char_arg1, arg2)</td>
<td>values of arg2 for a single run selected by char_arg1</td>
<td></td>
</tr>
<tr>
<td>transpose(char_arg1, arg2)</td>
<td>transpose arg2 according to char_arg1</td>
<td></td>
</tr>
<tr>
<td>undef( )</td>
<td>undefined element</td>
<td></td>
</tr>
</tbody>
</table>

### Aggregation and moment operators for arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg(arg)</td>
<td>argument arithmetic mean of values</td>
</tr>
<tr>
<td>avgg(arg)</td>
<td>argument geometric mean of values</td>
</tr>
<tr>
<td>avgh(arg)</td>
<td>argument harmonic mean of values</td>
</tr>
<tr>
<td>avgw(arg1, arg2)</td>
<td>argument weighted mean of values</td>
</tr>
<tr>
<td>count(char_arg1, arg2)</td>
<td>count number of values according to char_arg1</td>
</tr>
<tr>
<td>hgr(char_arg1, int_arg2, real_arg3, real_arg4, arg5)</td>
<td>argument histogram of values</td>
</tr>
<tr>
<td>max(arg)</td>
<td>argument maximum of values</td>
</tr>
<tr>
<td>maxprop(arg)</td>
<td>index of the element where the maximum is reached the first time</td>
</tr>
<tr>
<td>min(arg)</td>
<td>argument minimum of values</td>
</tr>
<tr>
<td>minprop(arg)</td>
<td>index of the element where the minimum is reached the first time</td>
</tr>
<tr>
<td>sum(arg)</td>
<td>argument sum of values</td>
</tr>
<tr>
<td>var(arg)</td>
<td>argument variance of values</td>
</tr>
</tbody>
</table>

### Multiple aggregation and moment operators for arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>max_n(arg1, ..., argn)</td>
<td>maximum per element</td>
</tr>
<tr>
<td>maxprop_n(arg1, ..., argn)</td>
<td>argument position (1 ... n) where the maximum is reached the first time</td>
</tr>
<tr>
<td>min_n(arg1, ..., argn)</td>
<td>minimum per element</td>
</tr>
<tr>
<td>minprop_n(arg1, ..., argn)</td>
<td>argument position (1 ... n) where the minimum is reached the first time</td>
</tr>
</tbody>
</table>

### Dimension related aggregation and moment operators for arguments

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg_l(char_arg1, arg2)</td>
<td>dimension related argument arithmetic means of values of arg2</td>
</tr>
<tr>
<td>avgg_l(char_arg1, arg2)</td>
<td>dimension related argument geometric means of values of arg2</td>
</tr>
<tr>
<td>avgh_l(char_arg1, arg2)</td>
<td>dimension related argument harmonic means of values of arg2</td>
</tr>
<tr>
<td>avgw_l(char_arg1, arg2, arg3)</td>
<td>dimension related argument weighted means of values of arg2</td>
</tr>
<tr>
<td>count_l(char_arg1, char_arg2, int_arg3)</td>
<td>dimension related count numbers of values of arg3</td>
</tr>
<tr>
<td>hgr_l(char_arg1, char_arg2, int_arg3, real_arg4, real_arg5, arg6)</td>
<td>dimension related argument histograms of values of arg6</td>
</tr>
<tr>
<td>max_l(char_arg1, arg2)</td>
<td>dimension related argument maxima of values of arg2</td>
</tr>
<tr>
<td>maxprop_l(char_arg1, arg2)</td>
<td>dimension related argument position (1 ... n) where the maximum of arg2 is reached the first time</td>
</tr>
<tr>
<td>min_l(char_arg1, arg2)</td>
<td>dimension related argument minima of values of arg2</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------------------------------------------------------</td>
</tr>
<tr>
<td>minprop_l(char_arg1,arg2)</td>
<td>dimension related argument position (1 ... n) where the minimum of arg2 is reached the first time</td>
</tr>
<tr>
<td>sum_l(char_arg1,arg2)</td>
<td>dimension related argument sums of values of arg2</td>
</tr>
<tr>
<td>var_l(char_arg1,arg2)</td>
<td>dimension related argument variances of values of arg2</td>
</tr>
</tbody>
</table>

**Multi-run operators (behavioural analysis)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>behav(char_arg1,arg2)</td>
<td>general purpose operator for navigating and aggregating arg2 in the experiment space</td>
<td>Tab. 8.10 on page 84</td>
</tr>
</tbody>
</table>

**Multi-run operators (Monte Carlo analysis and optimization)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg_e(arg)</td>
<td>run ensemble mean</td>
<td>Tab. 8.12 on page 88</td>
</tr>
<tr>
<td>avgg_e(arg)</td>
<td>run ensemble geometric mean</td>
<td>Tab. 8.9 on page 83</td>
</tr>
<tr>
<td>avgh_e(arg)</td>
<td>run ensemble harmonic mean</td>
<td></td>
</tr>
<tr>
<td>avgw_e(arg1,arg2)</td>
<td>run ensemble weighted mean</td>
<td></td>
</tr>
<tr>
<td>cnf(real_arg1,arg2)</td>
<td>positive distance of confidence line from mean avg_e(arg2)</td>
<td></td>
</tr>
<tr>
<td>cor(arg1,arg2)</td>
<td>correlation coefficient between arg1 and arg2</td>
<td></td>
</tr>
<tr>
<td>count_e(char_arg1,arg2)</td>
<td>run ensemble count number of values</td>
<td></td>
</tr>
<tr>
<td>cov(arg1,arg2)</td>
<td>covariance between arg1 and arg2</td>
<td></td>
</tr>
<tr>
<td>ens(arg)</td>
<td>whole Monte Carlo run ensemble</td>
<td></td>
</tr>
<tr>
<td>hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)</td>
<td>heuristic probability density function</td>
<td></td>
</tr>
<tr>
<td>krt(arg)</td>
<td>kurtosis (4th moment)</td>
<td></td>
</tr>
<tr>
<td>max_e(arg)</td>
<td>run ensemble maximum</td>
<td></td>
</tr>
<tr>
<td>maxprop_e(arg)</td>
<td>run number where the maximum is reached the first time</td>
<td></td>
</tr>
<tr>
<td>med(arg)</td>
<td>median</td>
<td></td>
</tr>
<tr>
<td>min_e(arg)</td>
<td>run ensemble minimum</td>
<td></td>
</tr>
<tr>
<td>minprop_e(arg)</td>
<td>run number where the minimum is reached the first time</td>
<td></td>
</tr>
<tr>
<td>qnt(real_arg1,arg2)</td>
<td>quantile of arg2</td>
<td></td>
</tr>
<tr>
<td>reg(real_arg1,arg2)</td>
<td>linear regression coefficient to forecast arg2 from arg1</td>
<td></td>
</tr>
<tr>
<td>rng(arg)</td>
<td>range = max_e(arg) - min_e(arg)</td>
<td></td>
</tr>
<tr>
<td>skw(arg)</td>
<td>skewness (3rd moment)</td>
<td></td>
</tr>
<tr>
<td>stat_full(real_arg1,real_arg2, real_arg3,real_arg4,arg5)</td>
<td>full basic statistical measures</td>
<td></td>
</tr>
<tr>
<td>stat_red(real_arg1,real_arg2, arg3)</td>
<td>reduced basic statistical measures</td>
<td></td>
</tr>
<tr>
<td>sum_e(arg)</td>
<td>run ensemble sum</td>
<td></td>
</tr>
<tr>
<td>var_e(arg)</td>
<td>run ensemble variance</td>
<td></td>
</tr>
</tbody>
</table>

**Multi-run operators (local sensitivity analysis)**

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>See</th>
</tr>
</thead>
<tbody>
<tr>
<td>lin_abs(char_arg1,arg2)</td>
<td>absolute linearity measure</td>
<td>Tab. 8.13 on page 90</td>
</tr>
<tr>
<td>lin_rel(char_arg1,arg2)</td>
<td>relative linearity measure</td>
<td></td>
</tr>
<tr>
<td>sens_abs(char_arg1,arg2)</td>
<td>absolute sensitivity measure</td>
<td></td>
</tr>
<tr>
<td>sens_rel(char_arg1,arg2)</td>
<td>relative sensitivity measure</td>
<td></td>
</tr>
<tr>
<td>sym_abs(char_arg1,arg2)</td>
<td>absolute symmetry measure</td>
<td></td>
</tr>
<tr>
<td>sym_rel(char_arg1,arg2)</td>
<td>relative symmetry measure</td>
<td></td>
</tr>
</tbody>
</table>

**Tab. 15.8**  
Experiment post-processor built-in operators (in thematic order)
### 15.5.2 Experiment Post-Processor Built-In Operators (in Alphabetic Order)

- **arg**: general numerical argument
- **int_arg**: integer constant argument \( \geq 0 \)
- **real_arg**: real (float) constant argument
- **char_arg**: character argument

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Type</th>
<th>See</th>
<th>At page</th>
</tr>
</thead>
<tbody>
<tr>
<td>arg1 + arg2</td>
<td>addition</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>arg1 - arg2</td>
<td>subtraction</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>arg1 * arg2</td>
<td>multiplication</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>arg1 / arg2</td>
<td>division</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>arg1 ** arg2</td>
<td>exponentiation</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>+ arg</td>
<td>identity</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>- arg</td>
<td>negation</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>( arg )</td>
<td>parentheses</td>
<td>elemental</td>
<td>Tab. 8.3</td>
<td>70</td>
</tr>
<tr>
<td>abs(arg)</td>
<td>absolute value</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>acos(arg)</td>
<td>arc cosine</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>acot(arg)</td>
<td>arc cotangent</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>asin(arg)</td>
<td>arc sine</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>atan(arg)</td>
<td>arc tangent</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>avg(arg)</td>
<td>argument arithmetic mean of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>avg_e(arg)</td>
<td>run ensemble mean</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>avg_l(char_arg1,arg2)</td>
<td>dimension related argument arithmetic means of values of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>avgg(arg)</td>
<td>argument geometric mean of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>avgg_e(arg)</td>
<td>run ensemble geometric mean</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>73</td>
</tr>
<tr>
<td>avgg_l(char_arg1,arg2)</td>
<td>dimension related argument geometric means of values of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>avgh(arg)</td>
<td>argument harmonic mean of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>avgh_e(arg)</td>
<td>run ensemble harmonic mean</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>73</td>
</tr>
<tr>
<td>avgh_l(char_arg1,arg2)</td>
<td>dimension related argument harmonic means of values of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>avgw(arg1,arg2)</td>
<td>argument weighted mean of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>avgw_e(arg1,arg2)</td>
<td>run ensemble weighted mean</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>73</td>
</tr>
<tr>
<td>avgw_l(char_arg1,arg2, _arg3)</td>
<td>dimension related argument weighted means of values of arg3</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>behav(char_arg1,arg2)</td>
<td>general purpose operator for navigating and aggregating of arg2 in the experiment space</td>
<td>behav.</td>
<td>Tab. 8.10</td>
<td>84</td>
</tr>
<tr>
<td>classify(int_arg1,real_arg2, _real_arg3,real_arg4)</td>
<td>classification of arg4 into int_arg1 classes</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>clip(char_arg1,arg2)</td>
<td>clip arg2 according to char_arg1</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>cnf(real_arg1,arg2)</td>
<td>positive distance of confidence line from mean avg_e(arg2)</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>83</td>
</tr>
<tr>
<td>cor(arg1,arg2)</td>
<td>correlation coefficient between arg1 and arg2</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>cos(arg)</td>
<td>cosine</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>cosh(arg)</td>
<td>hyperbolic cosine</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>cot(arg)</td>
<td>cotangent</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>coth(arg)</td>
<td>hyperbolic cotangent</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>count(char_arg1,arg2)</td>
<td>count number of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>count_e(char_arg1,arg2)</td>
<td>run ensemble count</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Type</td>
<td>See</td>
<td>At page</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>------</td>
<td>-----</td>
<td>--------</td>
</tr>
<tr>
<td>count_l(char_arg1, char_arg2,char_arg3)</td>
<td>dimension related count numbers of values of arg3</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>cov(arg1,arg2)</td>
<td>covariance between arg1 and arg2</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>cumul(char_arg1,arg2)</td>
<td>cumulates arg2 according to char_arg1</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>dim(arg1,arg2)</td>
<td>positive difference</td>
<td>basic</td>
<td></td>
<td>71</td>
</tr>
<tr>
<td>ens(arg)</td>
<td>whole Monte Carlo run ensemble</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>exp(arg)</td>
<td>exponential function</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>flip(char_arg1,arg2)</td>
<td>flip arg2 according to char_arg1</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>get_data(char_arg1, char_arg2,char_arg3,arg4)</td>
<td>get data from an external file</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>get_experiment(char_arg1, char_arg2,char_arg3,arg4)</td>
<td>include an other experiment</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>get_table_fct(char_arg1, arg2)</td>
<td>table function with linear interpolation of table char_arg1 for position arg2</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>hgr(char_arg1,int_arg2, real_arg3,real_arg4,arg5)</td>
<td>argument histogram of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>hgr_e(char_arg1,int_arg2, real_arg3,real_arg4,arg5)</td>
<td>heuristic probability density function</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>hgr_l(char_arg1,char_arg2, int_arg3,real_arg4, real_arg5,arg6)</td>
<td>dimension related argument histograms of values of arg6</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>if(char_arg1,arg2,arg3,arg4)</td>
<td>general purpose conditional if-construct</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>int(arg)</td>
<td>truncation value</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>krt(arg)</td>
<td>kurtosis (4th moment)</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>lin_abs(char_arg1,arg2)</td>
<td>absolute linearity measure</td>
<td>sensitivity</td>
<td>Tab. 8.13</td>
<td>90</td>
</tr>
<tr>
<td>lin_rel(char_arg1,arg2)</td>
<td>relative linearity measure</td>
<td>sensitivity</td>
<td>Tab. 8.13</td>
<td>90</td>
</tr>
<tr>
<td>log(arg)</td>
<td>natural logarithm</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>log10(arg)</td>
<td>decade logarithm</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>mask(char_arg1,arg2,arg3)</td>
<td>mask elements of argument arg2</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>matmul(arg1,arg2)</td>
<td>matrix multiplication</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>max(arg)</td>
<td>argument maximum of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>max_e(arg)</td>
<td>run ensemble maximum</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>max_l(char_arg1,arg2)</td>
<td>dimension related argument maxima of values of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>max_n(arg1 ,..., argn)</td>
<td>maximum per element</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>maxprop(arg)</td>
<td>index of the element where the maximum is reached the first time</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>maxprop_e(arg)</td>
<td>run number where the maximum is reached the first time</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>83</td>
</tr>
<tr>
<td>maxprop_l(char_arg1,arg2)</td>
<td>dimension related argument position (1 ... n) where the maximum is reached the first time of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>maxprop_n(arg1 ,..., argn)</td>
<td>argument position (1 ... n) where the maximum is reached the first time</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>med(arg)</td>
<td>median</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>min(arg)</td>
<td>argument minimum of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>min_e(arg)</td>
<td>run ensemble minimum</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>min_l(char_arg1,arg2)</td>
<td>dimension related argument minima of values of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>min_n(arg1 ,..., argn)</td>
<td>minimum per element</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>minprop(arg)</td>
<td>index of the element where the minimum is reached the first time</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>Name</td>
<td>Meaning</td>
<td>Type</td>
<td>See</td>
<td>At page</td>
</tr>
<tr>
<td>--------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>-----------</td>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>minprop_e(arg)</td>
<td>run number where the minimum is reached the first time</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>minprop_l(char_arg1, arg2)</td>
<td>dimension related argument position (1 ... n) where the minimum is reached the first time of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>minprop_n(arg1 ,..., argn)</td>
<td>argument position (1 ... n) where the minimum is reached the first time</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>mod(arg1, arg2)</td>
<td>remainder</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>move_avg(char_arg1, char_arg2, int_arg3, arg4)</td>
<td>moving average of running length int_arg3 for arg4</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>nint(arg)</td>
<td>round value</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>nr_of_runs()</td>
<td>number of single runs of the current experiment</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>qnt(real_arg1, arg2)</td>
<td>quantile of arg2</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>rank(char_arg1, arg2)</td>
<td>rank of arg2 according to char_arg1</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>reg(arg1, arg2)</td>
<td>linear regression coefficient to forecast arg2 from arg1</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>regrid(char_arg1, arg2)</td>
<td>assign new coordinates to arg2</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>rng(arg)</td>
<td>range = max_e(arg) - min_e(arg)</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>run(char_arg1, arg2)</td>
<td>values of arg2 for a single run selected by char_arg1</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>sens_abs(char_arg1, arg2)</td>
<td>absolute sensitivity measure</td>
<td>sensitivity</td>
<td>Tab. 8.13</td>
<td>90</td>
</tr>
<tr>
<td>sens_rel(char_arg1, arg2)</td>
<td>relative sensitivity measure</td>
<td>sensitivity</td>
<td>Tab. 8.13</td>
<td>90</td>
</tr>
<tr>
<td>sign(arg)</td>
<td>sign of value</td>
<td>basic</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>sinh(arg)</td>
<td>hyperbolic sine</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>skw(arg)</td>
<td>skewness (3rd moment)</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>sqrt(arg)</td>
<td>square root</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>stat_full(real_arg1, real_arg2, real_arg3, real_arg4, arg5)</td>
<td>full basic statistical measures</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>stat_red(real_arg1, real_arg2, arg3)</td>
<td>reduced basic statistical measures</td>
<td>Monte C.</td>
<td>Tab. 8.12</td>
<td>88</td>
</tr>
<tr>
<td>sum(arg)</td>
<td>argument sum of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>sum_e(arg)</td>
<td>run ensemble sum</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>sum_l(char_arg1, arg2)</td>
<td>dimension related argument sums of values of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
<tr>
<td>sym_abs(char_arg1, arg2)</td>
<td>absolute symmetry measure</td>
<td>sensitivity</td>
<td>Tab. 8.13</td>
<td>90</td>
</tr>
<tr>
<td>sym_rel(char_arg1, arg2)</td>
<td>relative symmetry measure</td>
<td>sensitivity</td>
<td>Tab. 8.13</td>
<td>90</td>
</tr>
<tr>
<td>tan(arg)</td>
<td>tangent</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>tanh(arg)</td>
<td>hyperbolic tangent</td>
<td>trigonom.</td>
<td>Tab. 8.4</td>
<td>71</td>
</tr>
<tr>
<td>transpose(char_arg1, arg2)</td>
<td>transpose arg2 according to char_arg1</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>undef()</td>
<td>undefined element</td>
<td>advanced</td>
<td>Tab. 8.8</td>
<td>77</td>
</tr>
<tr>
<td>var(arg)</td>
<td>argument variance of values</td>
<td>aggr./mom.</td>
<td>Tab. 8.5</td>
<td>73</td>
</tr>
<tr>
<td>var_e(arg)</td>
<td>run ensemble variance</td>
<td>Monte C.</td>
<td>Tab. 8.9</td>
<td>83</td>
</tr>
<tr>
<td>var_l(char_arg1, arg2)</td>
<td>dimension related argument variances of values of arg2</td>
<td>aggr./mom.</td>
<td>Tab. 8.7</td>
<td>74</td>
</tr>
</tbody>
</table>

Tab. 15.9  Experiment post-processor built-in operators (in alphabetical order)
### 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 can not have pre-defined character argument values.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Argument number</th>
<th>Argument value (without quotation marks, pre-defined values are case-insensitive)</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>avg_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>avgg_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>avgh_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>avgw_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>behav</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>clip</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>count</td>
<td>1</td>
<td>[ all</td>
<td>def</td>
</tr>
<tr>
<td>count_e</td>
<td>1</td>
<td>[ all</td>
<td>def</td>
</tr>
<tr>
<td>count_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>count_l</td>
<td>2</td>
<td>[ all</td>
<td>def</td>
</tr>
<tr>
<td>cumul</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>flip</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>get_data</td>
<td>1</td>
<td>ascii</td>
<td></td>
</tr>
<tr>
<td>get_data</td>
<td>2</td>
<td>&lt;directory&gt;/file_name</td>
<td>(*)</td>
</tr>
<tr>
<td>get_data</td>
<td>3</td>
<td>&lt;directory&gt;/file_name</td>
<td>(*)</td>
</tr>
<tr>
<td>get_experiment</td>
<td>1</td>
<td>&lt;directory&gt;</td>
<td></td>
</tr>
<tr>
<td>get_experiment</td>
<td>2</td>
<td>&lt;model&gt;</td>
<td></td>
</tr>
<tr>
<td>get_experiment</td>
<td>3</td>
<td>&lt;directory&gt;/file_name</td>
<td>(*)</td>
</tr>
<tr>
<td>get_table_fct</td>
<td>1</td>
<td>&lt;directory&gt;/file_name</td>
<td>(*)</td>
</tr>
<tr>
<td>hgr</td>
<td>1</td>
<td>[ bin_no</td>
<td>bin_mid ]</td>
</tr>
<tr>
<td>hgr_e</td>
<td>1</td>
<td>[ bin_no</td>
<td>bin_mid ]</td>
</tr>
<tr>
<td>hgr_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>hgr_l</td>
<td>2</td>
<td>[ bin_no</td>
<td>bin_mid ]</td>
</tr>
<tr>
<td>if</td>
<td>1</td>
<td>[ &lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>lin_abs</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>lin_rel</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>mask</td>
<td>1</td>
<td>[ &lt;</td>
<td>&lt;=</td>
</tr>
<tr>
<td>max_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>maxprop_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>min_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>minprop_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>move_avg</td>
<td>1</td>
<td>sequence of digits 1 to 9</td>
<td>(**)</td>
</tr>
<tr>
<td>move_avg</td>
<td>2</td>
<td>[ lin</td>
<td>exp ]</td>
</tr>
<tr>
<td>rank</td>
<td>1</td>
<td>[ tie_plain</td>
<td>tie_min</td>
</tr>
<tr>
<td>regrid</td>
<td>1</td>
<td>ascii</td>
<td></td>
</tr>
<tr>
<td>run</td>
<td>1</td>
<td>[ run number</td>
<td>not pre-defined ]</td>
</tr>
<tr>
<td>sens_abs</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>sens_rel</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>sum_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
<tr>
<td>sym_abs</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>sym_rel</td>
<td>1</td>
<td>(not pre-defined, case insensitive)</td>
<td>(*)</td>
</tr>
<tr>
<td>transpose</td>
<td>1</td>
<td>sequence of digits 1 to 9</td>
<td>(**)</td>
</tr>
<tr>
<td>var_l</td>
<td>1</td>
<td>sequence of digits 0 and 1</td>
<td>(**)</td>
</tr>
</tbody>
</table>

**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 with the dimensionality of the non-character and non-constant argument under investigation.
15.5.4 Constant Arguments of Experiment Post-Processor Built-In Operators

Tab. 15.11 summarises for built-in operators constant argument values.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Argument number</th>
<th>Argument type</th>
<th>Argument value restriction</th>
</tr>
</thead>
<tbody>
<tr>
<td>classify 1</td>
<td>1</td>
<td>int_arg</td>
<td>[0</td>
</tr>
<tr>
<td>classify 2</td>
<td>2</td>
<td>real_arg</td>
<td>arg2 = arg3 = 0.</td>
</tr>
<tr>
<td>classify 3</td>
<td>3</td>
<td>real_arg</td>
<td>arg2 &lt; arg3</td>
</tr>
<tr>
<td>cnf</td>
<td>1</td>
<td>real_arg</td>
<td>[0.001</td>
</tr>
<tr>
<td>hgr</td>
<td>2</td>
<td>int_arg</td>
<td>[0</td>
</tr>
<tr>
<td>hgr</td>
<td>3</td>
<td>real_arg</td>
<td>arg3 = arg4 = 0.</td>
</tr>
<tr>
<td>hgr</td>
<td>4</td>
<td>real_arg</td>
<td>arg3 &lt; arg4</td>
</tr>
<tr>
<td>hgr_e</td>
<td>2</td>
<td>int_arg</td>
<td>[0</td>
</tr>
<tr>
<td>hgr_e</td>
<td>3</td>
<td>real_arg</td>
<td>arg3 = arg4 = 0.</td>
</tr>
<tr>
<td>hgr_e</td>
<td>4</td>
<td>real_arg</td>
<td>arg3 &lt; arg4</td>
</tr>
<tr>
<td>hgr_l</td>
<td>3</td>
<td>int_arg</td>
<td>[0</td>
</tr>
<tr>
<td>hgr_l</td>
<td>4</td>
<td>real_arg</td>
<td>arg4 = arg5 = 0.</td>
</tr>
<tr>
<td>hgr_l</td>
<td>5</td>
<td>real_arg</td>
<td>arg4 &lt; arg5</td>
</tr>
<tr>
<td>move_avg</td>
<td>3</td>
<td>int_arg</td>
<td>[0</td>
</tr>
<tr>
<td>stat_full</td>
<td>1</td>
<td>real_arg</td>
<td>[0.001</td>
</tr>
<tr>
<td>stat_full</td>
<td>2</td>
<td>real_arg</td>
<td>arg1 &lt; arg2</td>
</tr>
<tr>
<td>stat_full</td>
<td>3</td>
<td>real_arg</td>
<td>0. ≤ arg3 &lt; arg 4 ≤ 100.</td>
</tr>
<tr>
<td>stat_full</td>
<td>4</td>
<td>real_arg</td>
<td>arg1 &lt; arg2</td>
</tr>
<tr>
<td>stat_red</td>
<td>1</td>
<td>real_arg</td>
<td>[0.001</td>
</tr>
<tr>
<td>stat_red</td>
<td>2</td>
<td>real_arg</td>
<td>arg1 &lt; arg2</td>
</tr>
</tbody>
</table>

*Tab. 15.11  Constant arguments of experiment post-processor built-in operators*
15.6 Additionally Used Symbols for the Model and Operator Interface

Tab. 15.12 lists these symbols (subroutine, function and common block names) that are linked in addition to the SimEnv model interface functions in Tab. 5.5 from the object libraries $SE\_HOME/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) 998 and 999 are used.

<table>
<thead>
<tr>
<th>Used symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>csimenv_&lt;string&gt;</td>
</tr>
<tr>
<td>isimenv_&lt;string&gt;</td>
</tr>
<tr>
<td>jsimenv_&lt;string&gt;</td>
</tr>
<tr>
<td><em>nc</em>&lt;string&gt;</td>
</tr>
<tr>
<td>nf_&lt;string&gt;</td>
</tr>
<tr>
<td>c2f_dimids</td>
</tr>
<tr>
<td>cdf_routine_name</td>
</tr>
<tr>
<td>f2c_coords</td>
</tr>
<tr>
<td>f2c_counts</td>
</tr>
<tr>
<td>f2c_dimids</td>
</tr>
<tr>
<td>f2c_maps</td>
</tr>
<tr>
<td>f2c_strides</td>
</tr>
<tr>
<td>read_numrecs</td>
</tr>
<tr>
<td>write_numrecs</td>
</tr>
</tbody>
</table>

**Tab. 15.12** Additionally used symbols for the model interface

Tab. 15.13 lists these symbols (subroutine, function and common block names) that are linked in addition to the SimEnv operator interface functions in Tab. 8.16 and Tab. 8.17 from the object library $SE\_HOME/libsimenv.a to a user-defined experiment post-processing operator.

<table>
<thead>
<tr>
<th>Used symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>csimenv_&lt;string&gt;</td>
</tr>
<tr>
<td>isimenv_&lt;string&gt;</td>
</tr>
<tr>
<td>jsimenv_&lt;string&gt;</td>
</tr>
</tbody>
</table>

**Tab. 15.13** Additionally used symbols for the operator interface
15.7 Glossary

The glossary defines and/or explains terms in that sense they are used in this User Guide. An arrow $\rightarrow$ refers to another term in the glossary.

**Adjustment:** Numerical modification of a $\rightarrow$ target during an $\rightarrow$ experiment. Adjustments are related to an $\rightarrow$ experiment type and are described in the experiment description $\rightarrow$ user-defined file.

**ASCII:** The American Standard Code for Information and Interchange developed by the American National Standards Institute (http://www.ansi.org) is used in SimEnv to store information in $\rightarrow$ user-defined files and on request in post-processing output files.

**Behavioural analysis:** $\rightarrow$ Experiment type to inspect behaviour of a $\rightarrow$ model in a space, spanned up by $\rightarrow$ targets. The target space is scanned in a deterministic manner, applying pre-defined $\rightarrow$ adjustments of the targets with a flexible scanning strategy for target sub-spaces.

**Coordinate coord:** Each $\rightarrow$ dimension of a $\rightarrow$ variable and each $\rightarrow$ operand of an $\rightarrow$ operator in a $\rightarrow$ result with a $\rightarrow$ 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 with the $\rightarrow$ extent for this dimension. Consequently, each model output variable with a dimensionality greater than 0 resides at a assigned (multi-dimensional) $\rightarrow$ grid. Assignments for variables is done in the model output description $\rightarrow$ user-defined file.

**Coupling:** $\rightarrow$ model interface

**Data type:** The type of a $\rightarrow$ variable as declared in the $\rightarrow$ model and the corresponding model output description $\rightarrow$ user-defined file. SimEnv data types are byte, short, int, float, and double.

**Default value:** The nominal (standard) numerical value of an experiment $\rightarrow$ target. The default value is specified in the experiment description $\rightarrow$ user-defined file and for $\rightarrow$ the model interface at the language level also in the model code.

**Dimension:** $\rightarrow$ dimensionality

**Dimensionality dim:** The number of dimensions of a model $\rightarrow$ variable or of an $\rightarrow$ operator result in $\rightarrow$ experiment post-processing. In the model output description $\rightarrow$ user-defined file each variable a dimensionality is assigned to that corresponds with 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 $\rightarrow$ Unix / $\rightarrow$ Linux operating system commands stored in an $\rightarrow$ ASCII file. The sequence of operating system commands is directly interpreted and executed by a command line interpreter, the so-called shell. Contrary to $\rightarrow$ shell scripts a child shell is not spawned. A dot script is preceded by a dot and a space when calling it.

**Environment variable:** At $\rightarrow$ Unix / $\rightarrow$ 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 $\rightarrow$ model in a co-ordinated manner by applying $\rightarrow$ 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 $\rightarrow$ experiment. SimEnv post-processing enables navigation in the $\rightarrow$ target space that is sampled by an experiment as well as construction of additional output functions by declaration and computation of $\rightarrow$ results.

**Experiment post-processing operator:** $\rightarrow$ operator

**Experiment target:** $\rightarrow$ target
**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) → targets are assigned to an experiment type and experiment specific → adjustments and other information are assigned to the targets. Currently available experiment types are → behavioural analysis, → Monte Carlo analysis, → local sensitivity analysis, and → optimization.

**Extent ext:** The number of values for a dimension (from the → dimensionality) of a model → variable or of an → operator result in → experiment post-processing. Extents are always greater than 1. Model output variables and operator results of dimensionality 0 do not have an extent.

**Expression:** → result expression

**Fortran storage model:** A rule how to map the elements of a multi-dimensional data field to a 1-dimensional vector and vice versa. A multi-dimensional data field field(1:ext1, 1:ext2, ..., 1:extdim-1, 1:extdim) of → dimensionality dim and → extents ext1, ext2, ..., extdim-1, extdim is mapped in Fortran on a 1-dimensional data field vector(1:*ext1* *ext2* ... *extdim-1* *extdim) in the following way:

```fortran
ipointer = 0
do idim = 1 , extdim
  do i1 = 1 , ext1
    do i2 = 1 , ext2
      ipointer = ipointer + 1
      vector(ipointer) = field(i1 , i2 ,..., idim-1 , idim)
    enddo
  enddo
enddo
```

For a two-dimensional matrix this storage model corresponds to a column by column storage of the matrix to the vector, starting with the first column and for each column starting with the first row.

**GAMS:** The General Algebraic Modeling System (http://www.gams.com) is a high-level modeling system for mathematical programming problems. It consists of a language compiler and a stable 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.

**IEEE:** SimEnv can use on demand for storage of model and post-processor output the Institute of Electrical and Electronics Engineers (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 the SUSE-Linux implementation (http://www.suse.com) for Intel-based hardware and compatibles.

**Load Leveler:** The load leveler LoadL is a job management system that handles compute resources at IBM’s p655 cluster at PIK.

**Local sensitivity analysis:** → Experiment type with incremental → adjustments of → targets in the neighbourhood of the → default values of the targets. A local sensitivity analysis in SimEnv is always performed independently for all targets 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.

**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 (→ targets) into a sequence of output values...
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, and → GAMS. Additionally, models can be interfaced at the → shell script level by using shell script syntax elements. For all interface techniques the interfaced model itself has to be wrapped into a shell script.

Model output variable: → variable

Monte Carlo analysis: → Experiment type with pre-single run perturbations of experiment → targets. Each perturbed target a → probability density function pdf with function parameters is assigned to. During the → experiment → adjustments of the targets are realizations from the pdf’s using random number techniques. In → experiment post-processing statistical measures can be derived from model output of the run ensemble. A prominent statistical measure is the heuristic pdf (histogram) of a model → variable and its relation to the pdf’s of the targets.

NetCDF: Network Common Data Form 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 (http://www.unidata.ucar.edu). NetCDF is freely available. SimEnv follows for model and → experiment post-processing output storage the NetCDF Climate and Forecast (CF) metadata convention 1.0-beta4 (http://www.cgd.ucar.edu/cms/eaton/cf-metadata/index.html) and extends it.

OpenDX: The Open Data Explorer 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 → target, a constant, a character string, → a macro and an operator.

Operator: Computational algorithm how to transform the values of a sequence of → operands into the values of the operator result during → experiment post-processing. An operator transforms → dimensionality, → extents, and → coordinates from the operands into the corresponding information for the operator result. There are built-in elemental, basic, and advanced operators as well as built-in operators related to specific → experiment types. Additionally, SimEnv offers specification of user-defined operators according to an operator interface. User-defined operators are announced to the system in the operator description → user-defined file.

Optimization: → Experiment type to minimize a cost function (objective function) over a bounded → target space. In SimEnv a simulated annealing strategy (check Section 4.5 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 target space that fits best the model behaviour with respect to the reference data.

Parallel Operating Environment: → POE

POE: The Parallel Operating Environment POE on IBM’s p655 cluster at PIK supplies services to allocate nodes, assign jobs to nodes and launch jobs.

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.

Shell script: A sequence of → Unix / → Linux operating system commands stored in an → ASCII file. A shell script is interpreted and executed by a command line interpreter, the so-called 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.

Simulation: Performing → experiments with → models

Target: Element of the input set of a → model. Targets are manipulated numerically during an → experiment. Targets can be addressed in → experiment post-processing and they have there a → dimensionality of 0.

Target adjustment: → adjustment

Unix: A computer operating system (http://www.unix.org), originally developed at AT&T/USL. SimEnv runs under the AIX Unix implementation for RS6000 hardware and compatibles from IBM.

User-defined files: A set of → ASCII files to describe → model, → experiment, → operator, → macro, and → GAMS model specific information and to determine general SimEnv settings. All user-defined files follow the same syntax rules.

Variable: Element of the output set of a → model that is stored in a SimEnv model output format. Variables are defined in the model output description → user file and they are output from the model to SimEnv data structures. Each variable has a unique → data type, a → dimensionality, → extents and an assigned → grid. Normally, a variable consists of a series of values, forming a field.

White spaces: → ASCII characters space (blank) and horizontal tabulator used in → user-defined files or within result expressions in → experiment post-processing.

Workspace: The directory, a SimEnv service was started from.