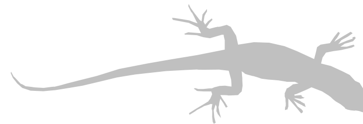
Eco-Hydrological Simulation Environment
(`echse`)

Documentation of Pre- & Post-Processors



Author	David Kneis
Affiliation	Institute of Earth and Environmental Sciences Hydrology & Climatology Section, University of Potsdam, Germany
Contact	david.kneis [at] uni-potsdam.de

Project	PROGRESS
Sub-project	D2.2
Funding	German Ministry of Education and Research (BMBF)
Last update	March 7, 2014

Please help to improve this document by sending suggestions, corrections, wishes, and other useful feedback to the author (see above).

Contents

1 Catchment modeling utilities (R-package `topocatch`) 7

- 1.1 Purpose . . . 7
- 1.2 Installation . . . 8
- 1.3 Standard documentation . . . 8
- 1.4 Supported data file formats . . . 8
  - 1.4.1 Overview . . . 8
  - 1.4.2 ASCII grid format . . . 8
  - 1.4.3 Shape file format . . . 9
- 1.5 Typical usage . . . 9
  - 1.5.1 Step 1: Filling of the elevation model (`dem.fill`) . . . 9
  - 1.5.2 Step 2: Analysis of the filled elevation model (`dem.analyze`) . . . 9
  - 1.5.3 Step 3: Identification of model objects (`hydroModelData`) . . . 10
  - 1.5.4 Step 4: Calculation of additional sub-basin attributes . . . 12
  - 1.5.5 Step 5: Estimation of river cross-section properties . . . 13
- 1.6 Practical hints . . . 16
  - 1.6.1 Processing of raster data . . . 16
  - 1.6.2 Making input grids consistent . . . 16
  - 1.6.3 Creating a proper shape file . . . 17
  - 1.6.4 Handling very short reaches . . . 17
  - 1.6.5 Special features in the river net . . . 18
  - 1.6.6 Extraction of river-cross sections from elevation models . . . 19
- 1.7 TODO . . . 19
  - 1.7.1 HRU support . . . 19

2 R-package for model optimization (`mops`) 21

- 2.1 Purpose . . . 21
- 2.2 Installation . . . 21
- 2.3 Standard documentation . . . 21
- 2.4 Theoretical background . . . 22
  - 2.4.1 Goal of optimization . . . 22
  - 2.4.2 Optimization methods . . . 22
  - 2.4.3 Model error as objective function . . . 22
  - 2.4.4 Semi-automatic calibration . . . 23
- 2.5 Important methods in **`mops`** . . . 24
  - 2.5.1 `update_template` . . . 24
  - 2.5.2 `modelError_multiDim` . . . 25
  - 2.5.3 `mcs_run` and `mcs_eval` . . . 25
- 2.6 Example: Monte-Carlo simulation . . . 25

2.6.1 Model equations . . . 25
2.6.2 Model implementation . . . 25
2.6.3 Observed data . . . 27
2.6.4 Monte-Carlo experiment . . . 28
2.7 Using **mops** with an **echse**-based model . . . 29
2.8 Troubleshooting . . . 31
2.8.1 General recommendations . . . 31
2.8.2 Specific recommendations . . . 32

3 Filling of gaps in meteorological time series (meteofill) 33
3.1 Purpose . . . 33
3.2 Methods . . . 34
3.2.1 Filling of gaps . . . 34
3.2.2 Inverse-distance approach . . . 35
3.2.3 Residual interpolation . . . 36
3.3 Arguments and invocation of **meteofill** . . . 37
3.4 Input . . . 38
3.4.1 Locations table . . . 38
3.4.2 Time series file . . . 39
3.5 Output . . . 39
3.6 Hints for practical usage . . . 39
3.6.1 Missing-only data (options beyond persistence) . . . 39

4 River cross-section analysis (xsAnalyzer) 41
4.1 Purpose . . . 41
4.2 Methods . . . 41
4.3 Arguments and invocation of **xsAnalyzer** . . . 42
4.4 Input . . . 43
4.4.1 Geometry data . . . 43
4.4.2 Flow values of interest . . . 43
4.4.3 Units . . . 43
4.5 Output . . . 45

5 Time series visualization tool (tsplot) 47
5.1 Purpose . . . 47
5.2 Required software . . . 47
5.3 Instructions files . . . 47
5.4 Expected format of data files . . . 48
5.5 Invoking **tsplot** . . . 49
5.5.1 On Linux . . . 49
5.5.2 On Windows . . . 49

List of figures 51

List of tables 53

Bibliography 54

Chapter 1

Catchment modeling utilities (R-package topocatch)

1.1 Purpose

The purpose of **topocatch** (Kneis, 2013) is to extract and pre-process information required by semi-distributed rainfall-runoff models from spatial data. This includes, for example

- the identification of (sub)-catchments.
- the determination of basic attributes of sub-catchments and river reaches.
- the analysis of input-output relation between the modeled objects (catchments, reaches, nodes, etc.).
- the estimation of river-cross section properties for sites where no survey data are available.

topocatch is distinguished from similar pre-processors by the following attributes:

Non-interactive Once a suitable R-script is written, the pre-processing runs without user interaction. Thus, whenever the spatial input data changes due to updates, corrections, or modifications, *all* steps of processing can be repeated without any effort.

Optional river net generation Pre-processors for hydrological models typically generate a river net from the digital elevation model (DEM). This option is also available in **topocatch**. As an alternative, however, the user can supply an existing river net file. Such a file may originate, for example, from digitized topographic maps, field surveys or it could be a DEM-derived river net including manual adjustments. Supplying an external (or manually modified) river net file results in great flexibility. Some advantages are:

- It can be achieved that the computed sub-catchments closely correspond to their natural counterparts.
- The user has the chance to split long reaches into several shorter segments and, hereby, gets control over both the minimum *and maximum* size of sub-catchments.
- Special features (such as reservoirs, canals, or pipes) may be integrated into the river net file and considered in the processing.

Created output The produced outputs are mostly in a format which can readily be used as input to rainfall-runoff models built with the **echse** simulation environment (see Kneis, 2012a).

Until 2012, **topocatch** used to be single, self-contained R-script whose behavior was controlled by a large number of configuration options. In 2013, **topocatch** was converted into a regular R-package to facilitate software maintenance, distribution, and documentation. Therefore, **topocatch** is no longer a single, ready-to-use script. Instead, the user has to write his/her own R-script that combines the various methods supplied by the package in a reasonable way. Some guidelines are provided in subsequent sections. Refer to the package's documentation for details on the various methods.

1.2 Installation

In order to use the **topocatch**, the **R** software for statistical computing is required. See Kneis (2012b) for information on how to obtain and install this software.

The **topocatch** package is distributed as a tarball archive. See the respective section in Kneis (2012b) for more information on how to install such add-on packages on your system. Note that **topocatch** internally uses Fortran source code. Thus, a Fortran compiler must be available on the system in order to successfully build the package. The recommended compiler is GNU **gfortran**.

Currently, **topocatch** depends on the following additional R-packages: **shapefiles**, **maptools**, and **foreign**. These packages need to be installed prior to **topocatch**.

1.3 Standard documentation

After the **topocatch** package has been loaded with the R command

```
library("topocatch")
```

a list of all provided methods can be generated with

```
help(package="topocatch")
```

The documentation of the individual methods can be displayed by typing the question mark followed by the name of the method, for example

```
?hydroModelData
```

Those who consider to use this package for the first time probably want to read the following sections to get a better understanding of the general concepts and the purpose of the various methods.

1.4 Supported data file formats

1.4.1 Overview

A typical pre-processing script for hydrological modeling using **topocatch** uses the following spatial data sets as input (see Fig. 1.1):

A digital elevation model (DEM) as a grid The grid values can be either integers or floating point numbers.

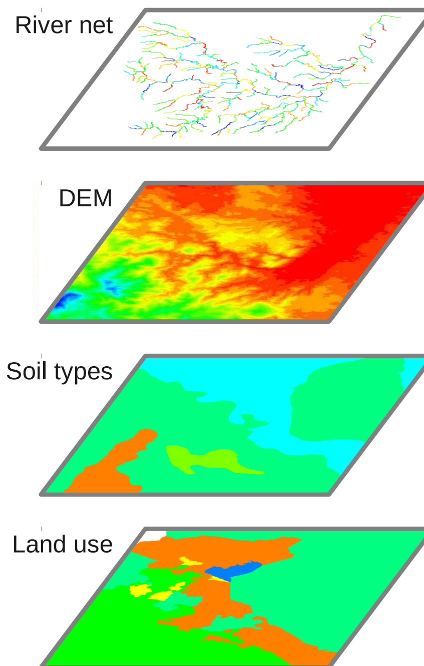


Figure 1.1: *The four spatial input data sets of* ***topocatch****.*

One or more soil maps as grid(s) The grid values can be integers (e. g. encoding the soil types) or quantitative properties such as conductivities, for example.

A map of land use The grid values are usually integers encoding the land use classes.

An optional vector file representing the river net Must be a shape file containing line features (sometimes called arcs). The attribute table must have (at least) one field with feature IDs (of integer type) and a class field (of type string). See Sec. 1.6.3 on practical issues and restrictions related to the creation of this file. If this input is not available, **topocatch** provides a method to generate an appropriate vector file from the DEM.

Thus, we deal with both raster and vector data.

1.4.2 ASCII grid format

All input raster data, e. g. those mentioned in Sec. 1.4.1, used by **topocatch** must be ASCII grids. This is a widely used exchange format for spatial raster data. It was originally used by ESRI's GIS systems but can be

```
ncols        6
nrows        5
xllcorner    3310800
yllcorner    5601975
cellsize     100
NODATA_value -9999
0 0 1 1 1 1
1 1 1 1 1 1
1 1 1 1 1 -9999
1 1 2 2 2 -9999
1 2 2 3 3 5
```

Figure 1.2: *Example of a file in ASCII grid format.*

imported and exported by many other GIS, including free software such as QGIS.

A file in ASCII grid format is a plain text file (Fig. 1.2). The first six lines of the file contain header information and the remaining lines hold the actual grid data as a matrix (one row per line). The meaning of the keywords in the header in as follows:

ncols Number of columns in the matrix of values starting at line 7 of the file.

nrows Number of rows in the matrix of values starting at line 7 of the file.

xllcorner X-coordinate corresponding to the lower left corner of the cell in the lower left corner of the matrix. Defines the Western border of the grid.

yllcorner Y-coordinate corresponding to the lower left corner of the cell in the lower left corner of the matrix. Defines the Southern border of the grid.

cellsize Length of a grid cell's edge. This is a single value, i.e. cells are quadratic.

NODATA_value Numerical value used to identify missing (or invalid) data in the values matrix.

1.4.3 Shape file format

This vector geo-data format can be im- and exported by most GIS systems including ESRI's GIS software, and QGIS, for example. It can be imported and exported by R as well. A shape file consists of (at least) three separate files with an identical basename and the extensions `.shp`, `.shx`, and `.dbf`. These are binary files containg coordinates, indices, and attribute data, respectively.

1.5 Typical usage

The methods provided by the **`topocatch`** package can be combined in various ways. The subsequent sections describe a typical case of usage based on the most important high-level routines.

1.5.1 Step 1: Filling of the elevation model (`dem.fill`)

Most digital elevation models (DEM) contain sinks. They represent either natural or man-made depressions in the earth surface or artifacts of remote sensing. The filling of such sinks is a necessary step because their existence would hinder the identification of continuous drainage paths. In contrast to other software tools, the designated method in **`topocatch`** uses a non-iterative approach to fill the sinks. Its method's name is `dem.fill`. See the package's internal documentation for details on the method's arguments.

For large elevation models, the filling of sinks may consume a considerable amout of computation time. Therefore, it may be favourable to apply `dem.fill` once and to save its output for later re-use. This is highly if the entire pre-processing script is still under development.

1.5.2 Step 2: Analysis of the filled elevation model (`dem.analyze`)

The method `dem.analyze` is used to analyze the sink-filled elevation model. It computes flow direction codes, the flow accumulation, the concentration time index and it finally generates a vector file of the flow paths. See the package's internal documentation for details on the method's arguments. Some details on internal algorithms are given below.

Identification of flow directions

The flow direction is computed for each cell of the DEM. It is identical to the direction of the steepest downward gradient (single-direction approach). The used algorithm is capable of handling locally flat areas. The result is an integer grid with values in the range 1...8. The meaning of these codes is shown in Fig. 1.3.

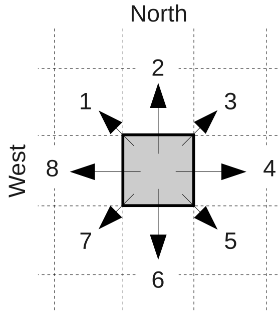


Figure 1.3: Flow directions encoded as integer values.

Calculation of flow accumulation

For each cell of the DEM, the flow accumulation is computed as the number of upstream cells. This information is obtained from the grid of flow direction codes.

Calculation of concentration time indexes

A concentration time index cti is computed for each raster cell. The cti is derived from the definition of the flow velocity u according to Eqn. 1.1 and Manning's equation (Eqn. 1.2).

$$u = \frac{L}{T} \tag{1.1}$$

$$u = \frac{1}{n} \cdot \sqrt{S_0} \cdot R^{2/3} \tag{1.2}$$

In Eqn. 1.1, L is the length of the flow path and T is the corresponding travel time. In Eqn. 1.2, n represents the roughness (known as Manning's n), S_0 is the energy slope, and R is the hydraulic radius. For steady flow problems, S_0 is equivalent to the surface slope. For small flow depths (in particular for overland flow), R is dominated by the flow width and the flow depth has little influence. Merging Eqns. 1.1 & 1.2 into a single expression and solving for the travel time T yields Eqn. 1.3.

$$T = \frac{L}{1/n \cdot \sqrt{S_0} \cdot R^{2/3}} \tag{1.3}$$

The cti is finally derived from Eqn. 1.3 by simply neglecting the linear terms $1/n$ and $R^{2/3}$. This is equivalent to treating $n/R^{2/3}$ as a scaling constant for which a value of 1 is assumed. Then, the cti for a particular cell k of the elevation model can be calculated with Eqn. 1.4.

$$cti(k) = \sum_{i=1}^{nk} \frac{L_i}{\sqrt{dz_i/L_i}} \tag{1.4}$$

In this equation, nk represents the number of raster cells through which the runoff generated in cell k must flow until it is discharged into a river. L_i is the (horizontal) length of path segment i. If the flow direction code of cell i is 2, 4, 6, or 8 (see Fig. 1.3), L_i is equivalent to the width of a raster cell. If the flow direction code is an odd number, L_i is the width of a cell multiplied by $\sqrt{2}$. Furthermore, dz_i is the elevation difference between cell i and the neighboring cell into which cell i drains. The user must specify a lower limit for dz_i to avoid zero division in cases where a flow path crosses a flat area.

In the resulting grid, the cti value of a cell will be larger, the longer the flow path and the lower the surface slope along the flow path is. Small cti values are found for near-river cells and where steep surface slopes occur. Later in the processing, characteristic values (such as a mean cti) are computed for the individual catchments. These values may be used as indicators for the rate of runoff concentration in the individual catchments, since they integrate information on the catchment's shape, drainage density, and slope. The indicators primarily reflect the *differences* between the catchments in terms of runoff concentration. To estimate actual concentration times, the cti values need to be multiplied by appropriate calibration parameters (recall the derivation of Eqn. 1.4 from Eqn. 1.3).

Generation of drainage lines (river net)

The `dem.analyze` method finally generates a shape file representing the drainage lines, i. e.rivers, based on the computed flow direction codes. Since the elevation model is the only source of information used, the result may differ from reality. This is especially true where the drainage network was altered by human action (canals, reservoirs, river training, etc.).

1.5.3 Step 3: Identification of model objects (`hydroModelData`)

After the DEM has been pre-processed and analyzed by `dem.fill` and `dem.analyze`, the major ob-

jects used in hydrological modeling (sub-basins, river reaches, etc.) need to be identified. This is achieved by a call to the high-level method `hydroModelData`. See the package's internal documentation for details on the method's arguments. Some background on the internal algorithms is provided below.

Vector-to-raster conversion of drainage lines

For the identification of sub-basins, the drainage lines must be converted to a grid. The drainage lines must be provided as a shape file which can be created in the following ways:

- It can be generated by `dem.analyze` which is the usual option for catchments without human alteration of the drainage system.
- It can an output of `dem.analyze` with subsequent manual modifications. Such modifications are typically required to take special objects like reservoirs, for example, into account.
- The shape file can be a digitized version of the basin's true river net. In this way it can be achieved that the generated sub-basins correspond to the actual river net as close as possible.

The result grid is of type integer and has the same extent and resolution as the DEM. The value of grid cells touched by a particular line feature is set to the corresponding value of the shape file's ID field. A special 'nodata' value is assigned to all grid cells not touched by any line feature. Another special 'conflict' value is assigned to cells which are touched by several lines (with different IDs). After this step, a buffer (having the width of a certain number of cells) is created around the gridded lines (see Fig. 1.4).

The result grid is used later to initialize the computation of catchments. Note that line features to which no catchment should be assigned are excluded from the vector-to-raster conversion. The IDs of those features, therefore, do not appear in the result grid. Whether a catchment is generated for a particular line feature is mainly controlled by the entry in the class field of the shape file's attribute table.

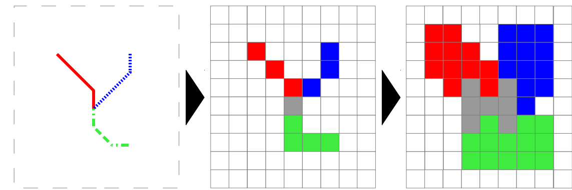


Figure 1.4: *Steps of the vector-to-raster conversion. Left: Line features as vectors data. Middle: Gridded line features. Right: Gridded lines after buffering using a width of 1 cell. The 'nodata' value is indicated by white color, the 'confict' value by gray color.*

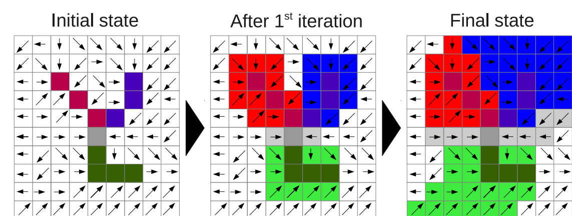


Figure 1.5: *The approach used to iteratively build the catchments. The dark-colored cells represent the initial cells carrying the ID of the corresponding reach (a buffer was omitted in this example). The arrows indicate the flow directions. In each iteration, new cells (in lighter colors) are added to the individual catchments. The gray-colored cell in the left grid has the special 'conflict' code because the junction of the reaches is located in that cell. A catchment is assigned to this cell too but the cells in this catchment carry a special 'conflict' value instead of a valid ID. These cells are reallocated (i. e. added to the surrounding catchments) in a later step of processing.*

Building of catchments

The catchments are determined based on the grid of the flow directions and the grid of the rasterized lines (Fig. 1.4). The latter grid is used as an initial estimate of the catchments where only near-river cells are set (to the ID of the corresponding reach). Starting from these reach cells (or near-reach cells if a buffer is used), the individual catchments 'grow' step-by-step by adding cells which discharge into the already set cells (according to the flow direction grid). The procedure continues until the catchments have reached their final size, i. e. there are no more cells to add. The approach is illustrated in Fig. 1.5.

As illustrated in Fig. 1.5, cells discharging into a 'conflict' cell (i. e. a cell that is in contact with multiple reaches), cannot be assigned to the catchment of a particular reach. Those cells are currently filled by a simple nearest neighbor interpolation.

Hydrological network assembly

In this step, the linkage of the different objects (usually reaches and catchments) is analyzed and information on

input-output-relations are dumped to tables. These tables can directly be used as an input for object-based hydrological catchment models built with the **echse** software.

The analysis of the network comprises a number of sub-steps, of which only the most important are mentioned below:

1. The data in the shape file of the river network are checked for errors (non-unique IDs, missing attributes, etc.).

2. The reach defining the systems outlet is identified.

3. For each line feature contained in the shape file, the downstream neighbor is determined. In the usual case, this means that for each reach object, the connected downstream reach is identified.

4. A catchment object is being linked to the line features (namely to all reach objects). Whether this actually happens for a particular line feature depends on the class of the feature (defined in the attribute table's class field) and further user-supplied settings.

5. The upstream neighbor(s), if any, are identified for each line feature. If the number of upstream neighbors is greater than 1, an appropriate node object is inserted. The function of the node object is to collect the information (usually inflow data) from all upstream neighbors and to provide this information to the receiving downstream object.

6. Several output tables are generated.

Finally, basic attributes of the major objects are computed. For sub-basins, the location of the center of gravity and the drained area are calculated. For river reaches the computed properties include

- the coordinates and elevation of end points,
- the reach length,
- the estimated bed slope,
- and the total area of upstream catchment.

As an optional step, the relation between the identified objects and stream gages can be identified. It is then known for each object whether a particular stream gage is affected by this object's output. Such information is helpful during the calibration of hydrological models. It also helps to spatially 'truncate' the model, i. e. to restrict the model domain to the catchment of a specific stream gage.

1.5.4 Step 4: Calculation of additional sub-basin attributes

Many hydrological catchment models require additional data in order to derive the sub-basin's parameters. Most typically, such information is obtained from digital maps of soil properties and land use.

For the purpose of data extraction from such maps the **topocatch** package provides the two methods `geogrid.zones.continuous` and `geogrid.zones.classified`. Both methods require as input

1. a primary grid with the computed sub-basins (defining the zones) and

2. a second grid with the data to be analyzed for each zone.

Extraction of continuous data

The `geogrid.zones.continuous` method assumes that the second input grid contains data on a spatially continuous variable like elevation, for example. For each individual zone (i. e. sub-basin) it calculates a statistics of that variable, namely the arithmetic mean, quantiles, and extremes with respect to the zone.

The method is typically used to determine for all sub-basins an average value and/or range of

- elevation,
- a quantitative soil property (depth, hydraulic conductivity, etc.),
- the concentration time index returned by the `dem.analyze` method.

Extraction of classified data

In contrast to that, the `geogrid.zones.classified` method assumes that the second input grid contains classified information such as land use codes (typically integers). For each individual zone (i. e. sub-basin) it calculates the areal shares of all the classes.

The method is typically used to determine the areal fractions of land use classes for all sub-basins.

1.5.5 Step 5: Estimation of river cross-section properties

Purpose

Hydrological catchment models usually require river cross-section (x-section) information to be available for all modeled river reaches. This is because the propagation and attenuation of flood waves is controlled by the river's bed slope as well as the cross-section's conveyance (hydraulic radius, wet area, and roughness).

In real-world river basins, x-section survey data are usually available for a very limited number of reaches only. Consequently, x-section data for all other reaches being part of the model domain need to be estimated. If estimates cannot be gathered from the elevation model (see Sec. 1.6.6), a spatial regionalization approach has to be adopted. The designated method in **`topocatch`** to perform this task is `xs.reachPars`. See R's internal help for detailed help on this methods's input arguments and outputs. Some additional background information is provided below.

Hydraulic properties of a single x-section

According to Manning's equation (Eqn. 1.5), the *steady* flow rate (Q) is controlled by four factors:

n The channel's roughness (friction and turbulence)

S_0 The slope of the channel.

R The hydraulic radius.

A The x-section's wet area.

$$Q = \frac{1}{n} \cdot \sqrt{S_0} \cdot R^{2/3} \cdot A \quad (1.5)$$

Eqn. 1.5 assumes that the x-section has a compact shape and, therefore, the channel's roughness can be described by a single value of n. In real x-sections, varying n values may be appropriate to describe the different resistance of the main channel and the flood plain (Cunge et al., 1980). However, the current version of **`topocatch`** is not capable of handling such so-called compound cross-sections and always assumes a unique n value [1].

[1] Compound cross-sections may be analyzed with the software described in Chap. 4

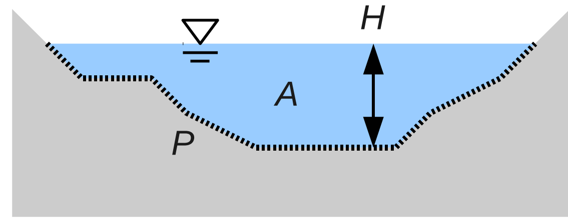


Figure 1.6: *Definition of basic properties of a river cross-section (D: Flow depth, A: Wet area, P: Wet perimeter). The hydraulic radius is $R = A/P$.*

The two factors A and R in Eqn. 1.5 are functions of the flow depth D (see Fig. 1.6). The shape of these functions is determined by the x-section's geometry. The steady flow rate Q is a function of D as well and $Q(D)$ represents the *rating curve*. From Eqn. 1.5 it is also obvious that there is a functional relation between Q and A. Considering that the storage volume V of a reach with length L equals $A \cdot L$, there are also functional relations $Q(V)$ and $V(Q)$, respectvely.

In **`topocatch`**, it is assumed that, for an individual x-section, the characteristic functions $A(D)$ and $R(D)$ can be approximated by simple power functions as in Eqn. 1.6 & 1.7. In these equations a, b, c, and d are empirical coefficients to be identified by least-squares fitting.

$$A(H) = a \cdot D^b \quad (1.6)$$

$$R(H) = c \cdot D^d \quad (1.7)$$

Cross-section estimation for arbitary sites

For reaches where x-section information is not available, a multi-step estimation procedure is applied by `xs.reachPars`. It aims at estimating the functions Eqns. 1.6 & 1.7 rather than the actual geometry data.

In the **first step**, two 'parent cross-sections' are identified from the pool of available survey data. These two cross-sections are selected in a way that

1. the parent cross-sections are located as close as possible to the reach of interest, and

2. the 1st cross-section has a smaller upstream catchment area and the 2nd one has a larger upstream catchment area than the reach of interest.

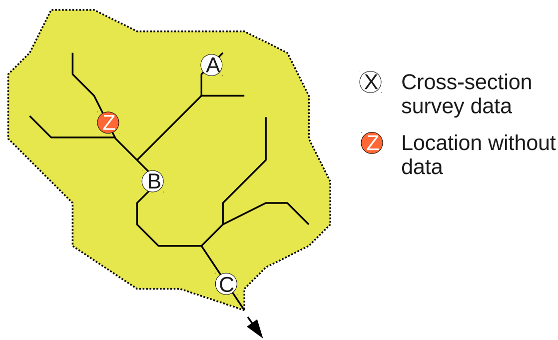


Figure 1.7: Selection of parent cross-sections for interpolation. In the example, the data from A and B would be used to estimate the x-section's properties at location Z.

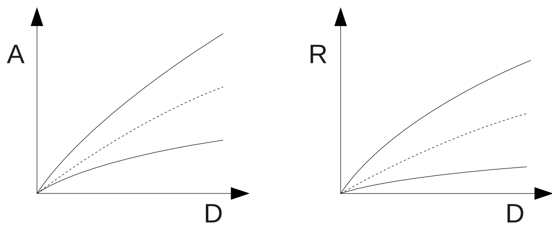


Figure 1.8: Interpolation of the cross-section's characteristic functions. Solid lines: Known relations for parent cross-sections. Dashed: Interpolated relation for a target location without survey geometry data.

This is illustrated in Fig. 1.7. Note that the two parent cross-sections are not necessarily located at the same branch of the river net.

In the **second step**, the characteristic functions $A(D)$ and $R(D)$ are computed for the two parent cross sections (solid graphs in Fig. 1.8).

Finally, in the **third step**, the characteristic functions $A(D)$ and $R(D)$ for the reach of interest are determined as a weighted average (dashed graphs in Fig. 1.8). The applied weights are derived from the upstream catchment areas. If, for example, the upstream catchment area of the two parent cross-sections was 5 and 10 km^2 and the reach of interest had an upstream catchment of 7 km^2, the information of the parent cross-sections would be weighted by $(7-5)/((7-5)+(10-7)) = 0.4$ and $(10-7)/((7-5)+(10-7)) = 0.6$, respectively. Note that, internally, the characteristic functions for all sites are approximated by power laws (Eqns. 1.6 & 1.7).

Once the functions $A(D)$ and $R(D)$ have been estimated for the reach of interest, all other hydraulic properties can be derived using Manning's equation and

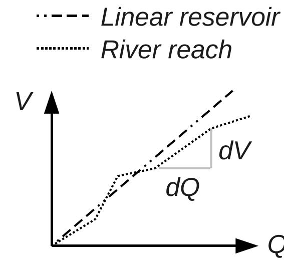


Figure 1.10: Relation between the flow rate Q and the storage volume V for a linear reservoir and a river reach with an irregularly shaped cross-section.

known values for the bed slope, the roughness, and the reach length.

It has to be noted that the above-mentioned approach of weighted averaging assumes that the x-sections's flow capacity (i. e. the values of A and R for a given flow depth D) are (positively) correlated with the size of the upstream catchment. In many cases, this appears to be a reasonable assumption. However, the correlation may be weak (or not exist at all) if

- the river basin's geology is heterogeneous, or
- a significant spatial gradient in rainfall is present.

In those cases, it may be better to sub-divide the river basin into zones of homogeneous geology/climate and to estimate the x-section characteristics separately for the zones.

It should also be noted that the assumed correlation between catchment size and flow capacity might not exist where cross-sections were constructed or altered by human action.

Example of the output

An example output of the `xs.reachPars` method is shown in Fig. 1.9. The data in that table can be used by hydrological models in various ways. Of special relevance are the values in the column dVdQ. They represent the derivative of the reach's storage volume with respect to the flow rate. This value (with the unit of a time) can be interepreted as the retention constant if the reach was treated as a (piece-wise) linear reservoir (Fig. 1.10).

```
# Hydraulic properties of a reach for STEADY UNIFORM flow
#   Object ID:      '365'
#   Upstream area: 632.7
#   Reach length:  34485.996
#   Bottom slope:  0.000812
#   Roughness:     30
# 1st / 2nd parent cross-section:
#   IDs:        'xsEstim_in.extractedXS.xs5015.txt' / 'xsEstim_in.extractedXS.xs5037.txt'
#   weights:    0.998 / 0.00156
#   dist.s :    214865.7 / 195208.3
#   up. areas: 627.39 / 4023.72
# Descr. of columns:
#   Q:    Stream flow (L/T)
#   D:    Normal depth, i.e. max. flow depth (L)
#   A:    Wet x-section area (L^2)
#   V:    Storage volume (L^3)
#   dVdQ: Est. derivative dV/dQ (T)
Q       D       A       V       dVdQ
0       0       0       0       96814.34
1       0.15    4.489   154817.373      96814.34
2       0.21    7.297   251631.715      91503.91
5       0.327   13.871  478349.54       69669.92
10      0.457   22.546  777510.074      56102.11
20      0.639   36.651  1263931.218     45967.99
50      0.995   69.66   2402299.489     34986.86
100     1.392   113.237 3905080.393     28179.45
200     1.946   184.069 6347792.32      23084.84
500     3.031   349.858 12065191.465    17571.48
1000    4.238   568.699 19612163.977    14151.79
2000    5.925   924.423 31879638.649    11593.46
```

Figure 1.9: *Example of an output file created by* `xs.reachPars`*.*

1.6 Practical hints

1.6.1 Processing of raster data

In addition to the routines mentioned in Sec. 1.5, the **`topocatch`** package provides many additional functions to facilitate the preparation of gridded spatial input data. For example:

- `geogrid.readAscii`: Reads spatial grids in ASCII format.
- `geogrid.writeAscii`: Outputs spatial grids in ASCII format.
- `geogrid.reclass`: For classification of numeric values in a grid.
- `geogrid.fillGaps`: Replaces missing cell values based on a nearest neighbor search.
- `geogrid.valuesAtPoints`: Extracts data from a grid for specified locations.
- `xs.extractDEM`: Extracts cross-sections from an elevation model.

See the R-package's built-in help for the full set of provided methods.

1.6.2 Making input grids consistent

Some of the methods contained in the **`topocatch`** package expect multiple grid as input arguments. This applies to the `geogrid.zones.continuous` and `geogrid.zones.classified` methods, for example. In those cases, the grids must cover (exactely) the same area with an identical resolution. In other words, the first 5 lines (see Fig. 1.4.2) of the corresponding ASCII grid files need to be identical. Since the data may come from different sources, a sequence of processing steps is typically required to make the grids spatially consistent. These steps are addressed in the subsequent paragraphs:

Resampling The first step is to make the cell size commensurate in all grids. This is typically known as resampling. For floating point grids like the DEM (bilinear) interpolation approaches are appropriate. When resampling integer grids (soil or land use data), one must use nearest neighbor methods, of course, in order not to introduce 'fractional' classes.

Clipping The second step is to clip the grids to an identical spatial window. The capabilities of clipping may be different in different GIS software. In QGIS 1.7.3, for example, one has to load the GDAL extension. With this extension loaded, a 'Clipper' functions is available in the menue 'Raster – Extraction' which is capable of clipping grid data. The spatial target window of the clip operation can be defined either by another layer or directly by specifying the coordinates of the target window's limits. For the Marikina catchment (Philippines) for example, the following target window was used:

Limit	Value	Defines border at
x1	289400	West
y1	1609000	South
x2	322200	East
y2	1641600	North

Conversion Once the grids have been clipped, they should be exported in ASCII grid format (see Sec. 1.4.2). Most GIS systems have this functionality. If this format is not supported, it may be possible to export the data in another matrix-based text format and then to manually add (or modify) the header (see Fig. 1.2). A typically used file extension for ASCII grid files is `.asc` but this is not a true standard.

Final check Last but not least, the exported ASCII grid files should be opened in a (powerful) text editor and the first 5 lines of the header (see Sec. 1.4.2) need to be compared.

Although the same target window (or layer defining that window) was used in the clipping operation, it may happen that the header information are not exactely identical. This is due to the fact that the clipping simply removes cells outside the target window. It does, however, not apply a real spatial transformation to the data (which is more difficult) and, therefore, cannot correct for spatial shifts in x- or y-direction being smaller than the extend of a single cell.

A possible solution would be to apply a real spatial transformation prior to clipping of the grids. A more convenient workaround is to simply substitute the headers in all grids by the header of one 'master grid' (the DEM, for example). This approach works as long as the first two lines in the original grid headers are identical (i.e. the grids have the same number of rows and columns). The simple copy & paste approach will in-

troduce a spatial shift in two of the grids but the error should be less than the extent of a single cell. This will be tolerable in most situations (considering that a spatial transformation is an approximation as well).

After this step, the header information in the grids should be identical *to the very last digit*, which is a prerequisite to run some of **topocatch**'s methods.

1.6.3 Creating a proper shape file

As mentioned in Sec. 1.4.1, a shape file of line features representing the river network is required as an input to the `hydroModelData` method. If this file is created manually rather than being generated by **topocatch**'s `dem.analyze` method, several aspects have to be considered related to both attributes and geometry.

Attributes The attribute table of the shape file must contain at least two fields: An ID field and a class field. The names of these fields can be chosen freely but recommended are 'id' and 'class', respectively.

The ID field must contain a *unique* value that identifies each single feature in the shape file, i. e. each single reach. The ID field should be of type integer.

The class field should be of string type. It allows to distinguish different classes of line features and to convey this information to **topocatch**. In a simple river system, a useful entry in the class field would be 'reach', for example, and this would be identical for all features. However, it would also be possible to include special features of a river network (such as reservoirs) in the shape file. For those features, a different class name (such as 'reservoir') would have to be entered in the class field.

Geometry and topology The conditions that must be fulfilled in terms of the features' geometry and topology are as follows:

Single-part lines only Some GIS systems support multi-part lines (line features with gaps in between). These must not appear in the shape file.

End-to-end connections Line features must always be connected end-to-end. Thus, a line must not end at an intermediate vertex of another line. This also implies that junctions are always formed by 3 (or more) individual lines whose ends share the same coordinates. The alternative model where a side branch ends at some intermediate point of the main branch is not supported (see Fig. 1.11 for an illustration).

Snapping Where two line features have a connection, the coordinates of the connected end points need to be *exactly* identical (to the very last digit). When digitizing the lines manually, this can *only* be achieved by using the so-called snap functionality provided by all GIS systems.

No loops The current version of **topocatch** only supports tree-like river systems. Thus, flow splits followed by junctions further downstream (i. e. loops) must not exist in the river net.

Minimum length of reaches As a rule of thumb, line features representing objects with a corresponding catchment – namely reaches – should not be shorter than about 4–5 times the cell size of the input grids. Thus, if the elevation model has a resolution of 100 m × 100 m, reaches should not be shorter than about 400–500 m. This is due to the fact that the shape file (vector data) is converted to a grid (raster data) within **topocatch**'s `hydroModelData` method and too short line features may get lost. Possible solutions for cases where the natural river system contains critically short reaches are discussed in Sec. 1.6.4. Note that line features to which no catchment should be assigned (which must be identified by an appropriate entry in the attribute table's class field) can be of arbitrary length.

Orientation of lines The orientation of the individual line features is of no importance. Thus, it is OK to digitize lines in upstream or downstream directions and mixing both directions in the shape file is OK too.

1.6.4 Handling very short reaches

In dense drainage networks, reaches may exist whose length is critically short (Fig. 1.12). Critical means that the length is less than about 4 times the resolution of the input grids (see Sec. 1.6.3). Those very short reaches may get lost during vector-to-raster conversion in the `hydroModelData` method and **topocatch** will generate an error because no catchment could be generated for some of the reaches contained in the shape file.

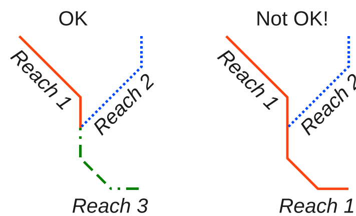


Figure 1.11: *Left: Proper junction formed by three individual reaches (identified by different colors and styles). Right: Improper junction with a tributary ending in the mid-section of another reach.*

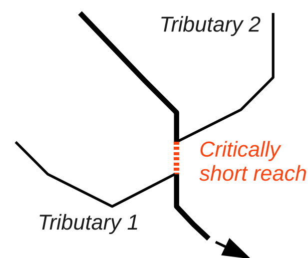


Figure 1.12: *Typical example of a critically short reach in a dense river network.*

There are several possible solutions to this problem:

Increasing the grid resolution By resampling the elevation model (and all other input grids) to a finer resolution, the reach lengths increases relative to cell size. Consequently, there is a higher chance that even short reaches are retained in the vector-to-raster conversion. The drawback of this approach is, however, that the input grids become much larger and the computation becomes slower. For example, reducing the cell size to 1/2 increases the number of values in the data matrix by a factor of 4.

Removal of the short reaches An alternative would be to edit the shape file and remove the critically short reaches. In the example shown in Fig. 1.12, this would mean that the short dashed line is deleted and the junctions up- and downstream of that reach are merged into an artificial junction with three inflows. The drawback of this approach is that manual work is necessary. Also, the shape file does no longer represent the actual network. Finally, if a very large number of short reaches is deleted, this may also lead to a systematic error in travel times.

Increasing the reach length The very short reaches could also be made longer manually in order to increase the probability of a successful vector-to-raster conversion. The disadvantages are similar to those related to the removal of the short reaches.

Using a separate class Another option which is recommended in most cases is to declare the short reaches not as reach objects but as instances of a different class. An appropriate name for this class might be something like 'shortReach' or 'link'. This is simply achieved by changing the entry in the attribute table's class field for the critically short reaches to 'shortReach' or 'link', respectively. Most GIS systems provide a suitable tool to do this sort of table column calculations using expressions. One can then inform **`topocatch`**'s `hydroModelData` method that no catchments should be build for objects of that special class. This essentially solves the problem, because (like all features not having a catchment), the short reaches are excluded from the vector-to-raster conversion. As a consequence, however, the newly introduced class must also be declared and implemented in the rainfall-runoff model. This class could either provide the same functionality as the normal reach class or just have the functionality of a link (which simply copies its input to its output). If the shape file is generated by **`topocatch`** itself (`dem.analyze` method), one may specify a critical reach length to automatically assign a different class name to short reaches.

1.6.5 Special features in the river net

As discussed earlier, the shape file may also contain other objects than river reaches. Such 'special' objects must

1. be represented as lines, even if the objects are actually punctual (like gages or control structures, for example), or have an areal extent (reservoirs, lakes, etc.). This is due to the nature of the shape file format which restricts the contents to a single feature class (points *or* lines *or* polygons).

2. be indicated by an appropriate entry in the attribute table's class field.

Furthermore, if a catchment should be assigned to these objects (as in the case of lakes, for example),

one needs to make **topocatch**'s `hydroModelData` method aware of this fact.

Example 1: A gage

In a standard hydrological model, one would probably not treat gages as model objects. One would rather simply let the model output the flow rate of the reach to which the gage is attached. In operational models, however, is may be useful to treat gages as objects, because then, a gage may have some functionality. Typically, the observed flow would be defined as an external input variable of gage objects. In addition, some user-defined rule would be implemented that controls whether the simulated *or* observed flow rates are submitted to the reach downstream of the gage.

In the shape file, a punctual gage object could be represented by a very short line feature (say of 1 m length). The resulting error in the system's total reach length would then be negligible.

Example 2: A reservoir

Some more effort is necessary to include objects with an areal extent into the shape file of line features. This is demonstrated in Fig. 1.13 with the example of a reservoir with two inflows. In situations like these, one must chose one line to represent the actual object. In the example, this is the line with ID 100. It is digitized in a zigzag manner to roughly cover the reservoir's surface area. This is done with the aim of increasing the chance that the reservoir's 'direct' catchment (i. e. the area draining to the reservoir's shore line) is properly estimated from the elevation model.

To establish the original inter-connection, an artificial object with ID 101 is introduced. It is defined as an object of a separate class, here named 'link'. The functionality of this class is to simply transfer data. In the example, this link object redirects the outflow of reach 1 to a single inflow location for the reservoir without introducing a time lag.

1.6.6 Extraction of river-cross sections from elevation models

In Sec. 1.5.5 it was mentioned that data on river cross-section geometries are required for the set-up of many hydrological models. Since survey data are usually scarce, it may be desireable to extract such

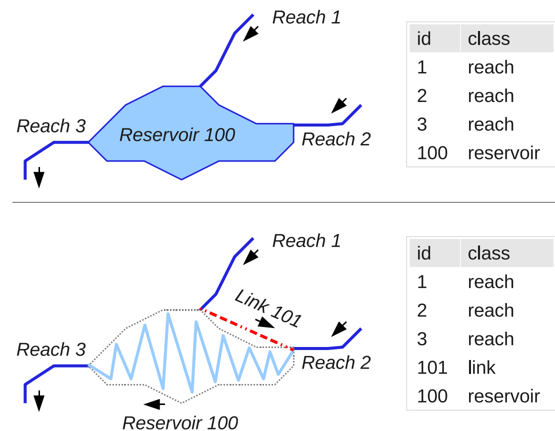


Figure 1.13: *Representation of a reservoir with two inflows in a map (top) and in the input shape file (bottom).*

data from digital elevation models. For that purpose, the **topocatch** package provides a method `xs.extractDEM`. See the R-package documentation for more information.

1.7 TODO

1.7.1 HRU support

The current version of **topocatch** does *not* generate the information needed by rainfall-runoff models using the hydrological reaction unit (HRU) approach. This is because of the fact that, so far, the **echse**-based hydrological models are intended to be used in operational forecasting. Such models typically need to be simpler than more process-oriented models for the sake of computational efficiency. However, it is not difficult to let **topocatch** generate input for HRU-based models and a future version might support this. **topocatch**'s source code would have to be adapted in two ways:

1. Soil and land use information need to be merged into a single map prior to the calculation of areal shares with respect to the sub-basins using the `geogrid.zones.classified` method. This could be done, for example, by multiplying the soil code by a factor of 1000 and then adding the land use code. Then, the code of a HRU consisting of soil type 5 and land use type 12 would be 005012.

2. In the `hydroModelData` method, an array of HRU-objects must be assigned to all reaches (and

possibly further objects). At present, a single catchment object is assigned only.

Chapter 2

R-package for model optimization (mops)

2.1 Purpose

The R-package **mops** provides a number of methods to facilitate the optimization of model parameters. The methods may be useful in the context of

- model calibration.
- updating of operational models.
- optimization studies.

The methods provided in the **mops** package are designed for the following conditions:

- The model, whose parameters are to be optimized, must be an executable file. In practice, this can be an executable built from source code (C++, FORTRAN, etc.) or a shell script. If it is a shell script, it typically calls an interpreter to process code written in a script language (R, Python, etc.).
- The executable must read the parameter values from plain text files. Thus, parameter values must *not* be read from binary files or passed as command line arguments.
- The input parameters to be optimized must be floating point numbers. Internally, of course, the model can apply any kind of type conversion (typically to integer or logical).
- The text files holding the parameter values may be of arbitrary structure. However, the executable must read the numbers *unformatted*. In other words, when reading a numeric value of one, for example, it must accept all of the following character representations `1`, `1.`, `1.0`, `1.0e+00`, `0.1e+01`. Note that legacy code sometimes uses *formatted* read statements and may be incompatible with **mops**.

echse-based models generally comply with all of the conditions listed above.

2.2 Installation

The **mops** package is provided as a tarball. The package file name is `mops_x.y.tar.gz` where x.y is a version number. See Kneis (2012b) for details on the installation procedure.

Note that the **mops** package depends on another R-package called **lhs** which allows for latin hypercube sampling. If the **lhs** package is not installed already, it needs to be downloaded and installed *prior* to installing **mops**.

2.3 Standard documentation

After the **mops** package has been loaded with the R command

```
library("mops")
```

a list of all provided methods can be generated with

```
help(package="mops")
```

The documentation of the individual methods can be displayed by typing the question mark followed by the name of the method, for example

```
?modelError_MCS
```

As always, the standard documentation for the methods is rather short. New users probably want to read

the following sections to get a better understanding of the general concepts and the dependencies between the methods in the **mops** package.

2.4 Theoretical background

2.4.1 Goal of optimization

Optimization generally refers to the minimization[1] of the value of a function whose value depends on one or more parameters. This function is called the *objective function*. Let's write the objective function as $f(p)$, with f being the function's name and p being a vector of parameters. In the special case of one-dimensional optimization, p is simply a vector of length 1.

The goal of *any* optimization procedure is to find a set of values for the parameters p that minimizes the value of f.

Example 1 (minimization, 1-dimensional): Determine the optimum dosage of a medicine for malaria protection that minimizes the total number of death from both infection and serious side effects (f: number of casualties, p: dosage).

Example 2 (maximization, 2-dimensional): Determine the amounts of fertilizer and water to maximize the yield of a tomatoe field (f: yield, p: amounts of fertilizer and water).

2.4.2 Optimization methods

Depending on the nature of f (non-linearities, discontinuities, local minima, etc.) and the number of unknown parameters in p, the optimization problem is more or less difficult to solve. Unfortunately, there is no general strategy that performs best in all cases. In fact, it is not even guaranteed that a *global* minimum of f is found at all.

One thing which is common to all optimization methods is that *multiple* calls to f for varying values of p are necessary (bold arrow in Fig. 2.1). Only then, the effect of a change in the parameter value(s) p on the value of f can be detected.

The basic strategies of optimization are distinguished by the way of how a new proposal for the values in p is generated, given the information from the previous calls

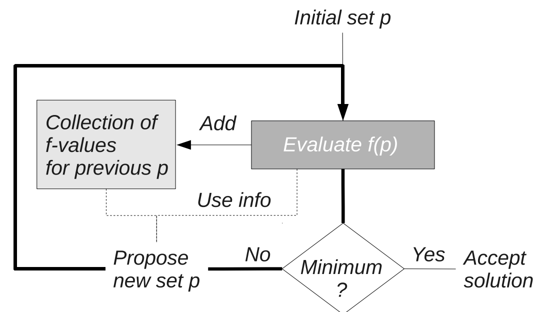


Figure 2.1: Basic outline of an optimization method.

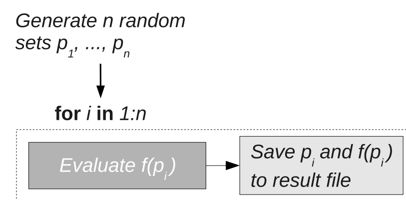


Figure 2.2: Basic outline of Monte-Carlo simulation.

to f for other values of p. The most important distinction is between *deterministic* and *stochastic* algorithms. Some examples and references for both types of strategies can be found in the help text of R's `optim` method.

As opposed to 'complete' optimization strategies, the technique of Monte-Carlo simulation does not identify a single optimum parameter set. Instead, the objective function is simply evaluated for a large number of random parameter sets (Fig. 2.2). It is then up to a human to inspect the output and to draw conclusions on the suitability of parameter values.

2.4.3 Model error as objective function

In the context of a dynamic simulation, the objective function measures the deviation between a simulated time series (produced by the model) and a corresponding time series of observations. Generally, the deviation (synonyms: model error, performance, goodness-of-fit) depends on the values of the model's parameters. It may be helpful to see that this is very similar to the common case of fitting a linear model to a set of x,y-data (Fig. 2.3). The typical differences are:

- In dynamic modeling, the x-axis represents the time.

[1] To solve a maximization problem, the objective function is simply multiplied by -1.

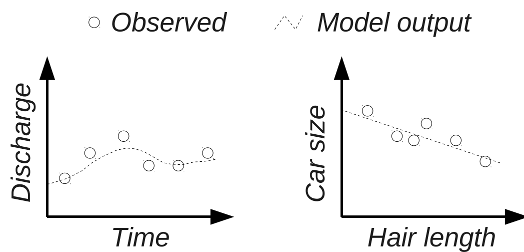


Figure 2.3: *Goodness-of-fit for a dynamic simulation model (left) and an empirical linear model (right).*

- Real-world dynamic models often have more parameters than the linear model which has only 2 (slope and intercept).
- In many dynamic models, the relations between parameter values and the output are non-linear.
- In the case of dynamic simulation models, numerical methods are required to determine a set of optimum parameters (see Sec. 2.4.2). For the linear model, an analytical expression exists to directly solve for slope and intercept.

The deviation between observations and the corresponding simulated values can be quantified by a variety of mathematical measures. In the case of linear fitting (Fig. 2.3, right), the sum of squared errors (SSE) is most often used. In dynamic modeling (Fig. 2.3, left), measures like the RMSE (root mean suared error) and the Nash-Sutcliffe index are usually preffered. They are based on the SSE as well but the values are more convenient to interpret.

In the context of dynamic modeling, the interior of a objective function $f(p)$ typically contains the steps listed in Table 2.1.

Table 2.1: *Interior of an objective function in the context of dynamic modeling. It is assumed that the model reads all parameter values from text files.*

#	Task
1	Get current parameter set p (passed via the funcion's argument list).
2	Update the model's input files with the current values in p.
3	Run the model.
4	Read the model output (a time series).
5	Read observed data.
6	Compute the model error from the data read in steps 4 & 5 using a suitable mathematical measure.
7	Return the result.

2.4.4 Semi-automatic calibration

Environmental simulation models often have a large number of conceptual parameters whose values have to be calibrated based on observations. Without having a deeper understanding of the model's functioning, i. e. the underlying equations, this is not a trivial task. Even if the equations are known, multi-dimensional, non-linear optimization remains a challenge and success is not guaranteed. The classic, deterministic optimization algorithms may fail for a number of reasons, such as truncation and round-off problems, insensitive parameters, complicated parameter interactions and compensations, non-continuous model behavior, or the existence of local minima. The alternative is the use of stochastic algorithms. Unfortunately, stochastic optimizers usually come with a number of algorithms parameters which affect convergence and computation times. If established general-purpose defaults do not exist, the estimation of these algorithm parameters is a problem on its own. Finally, a manual calibration by trial-and-error is often not practically feasible and the results have the reputation of being subjective.

Therefore, it is sometimes a good idea to fall back on the robust and straightforward concept of Monte-Carlo simulation (Fig. 2.2). Some advantages of Monte-Carlo simulation are:

- It simply cannot fail as long as the tested parameter sets do not cause invalid numeric results.
- There are no algorithm parameters.
- One can learn from the output about the (in)sensitivity of parameters, even if the model is a black box.

Especially useful is the concept of sequential Monte-Carlo simulation. This concept can be described by the following set of instructions:

a) Carry out a Monte-Carlo simulation with wide sampling ranges for all parameters.

b) Visualize the results from step a. For each varied parameter, a scatter plot should be created with the

parameter value on the x-axis and the model error on the y-axis.

c) Inspect all plots created in step b. If a structure is visible in a plot, indicating an optimum range for the particular parameter, modify the sampling range of that parameter. Usually, the sampling range can be chosen narrower (unless the initial range was not wide enough).

d) Return to step a, using the updated sampling range.

This semi-automatic approach to calibration has been successfully used in hydrological modeling (see, e. g. Kneis et al., 2012). It is certainly not among the most efficient strategies. However, the chance of complete failure is very low. Finally, with the above approach, it becomes visible whether the optimization problem is well behaved or of the nasty sort. Users of unsupervised optimization algorithms can only hope for a well behaved problem or guess on the cause of failure.

A practical example of parameter estimation using sequential Monte-Carlo simulation can be found in Sec. 2.6.4.

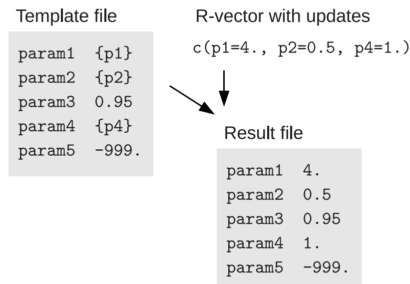


Figure 2.4: *Example application of the* `update_template` *method. Here, the characters to identify the start and the end of a placeholder are* "{" *and* "}"*, respectively.*

2.5 Important methods in mops

2.5.1 update_template

As illustrated in Figs. 2.1 and 2.2, optimization methods and Monte-Carlo simulation involve many successive of evaluations of the objective functions for different parameter values. In each evaluation, the model's input files need to be updated (recall Table 2.1). For this purpose, the **mops** package provides the `update_template` method.

The method takes a template file and a *named* vector of numerical values as input. It then scans the template file for the occurrence of *placeholders*. A placeholder is a string enclosed by two designated characters, typically some sort of fancy brackets like {}, [] or <>. The `update_template` method tries to replace every placeholder in the template file by the numerical value of a matching element from the input vector. A matching element is one whose name is identical to the placeholder's central string, i. e. the string between the designated characters. The mode of action of the `update_template` method is best demonstrated by the examples in Figs. 2.4 and 2.5.

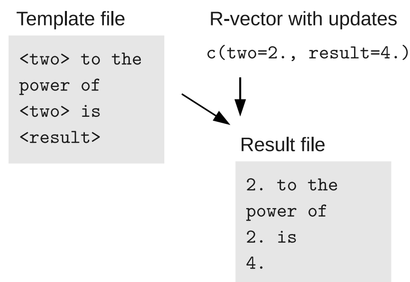


Figure 2.5: *Another example application of the* `update_template` *method. Here, the characters to identify the start and the end of a placeholder are* "<" *and* ">"*, respectively.*

Although the `update_template` can be called in isolation, it is mainly used as a sub-routine in other high-level methods of the **mops** package.

2.5.2 `modelError_multiDim`

The `modelError_multiDim` is the central high-level method of the **mops** package. It represents a complete objective function. For a given parameter set, the method is able to perform all the steps listed in Table 2.1. It can directly be used together with R's generic optimization method `optim`.

2.5.3 `mcs_run` and `mcs_eval`

The methods `mcs_run` and `mcs_eval` provide a complete framework for Monte-Carlo simulation. `mcs_run` generates the random parameter sets and carries out all the simulations. The function `mcs_eval` calculates the corresponding model errors and creates graphics to summarize the results of Monte-Carlo experiments. A complete example for the use of these methods is provided in Sec. 2.6.

2.6 Example: Monte-Carlo simulation

2.6.1 Model equations

This section introduces the practical use of the `mcs_run` method to perform a Monte-Carlo simulation. For the purpose of demonstration, we consider the fairly simple 1-dimensional advection-dispersion model (Eqn. 2.1). It describes the 1-dimensional (longitudinal) transport of dissolved, non-reactive matter in a river.

$$\frac{\partial c}{\partial t} = d \cdot \frac{\partial^2 c}{\partial x^2} - u \cdot \frac{\partial c}{\partial x} \tag{2.1}$$

c Concentration (g/m^3)
x River station (m)
d Dispersion coefficient in x-direction (m^2/s)
u Average flow velocity in x-direction (m/s)

This partial differential equation model has two parameters, d and u. For certain conditions, Eqn. 2.1 has the analytical solution presented in Eqn. 2.2.

$$c(x,t) = \frac{m}{A \cdot \sqrt{4 \cdot \pi \cdot d \cdot t}} \cdot exp\left(\frac{-(x-u \cdot t)^2}{4 \cdot d \cdot t}\right) \tag{2.2}$$

with the additional symbols
t Time (s)
m Mass (g)
A Wet cross-section area (m^2)

This solution assumes that

- The injection of mass at $t = 0$ occurs in an infinitesimally short time span.
- The mass is instantly distributed over the entire cross-section.
- The flow is steady (constant rate) and uniform (constant cross-section geometry).

The behavior of the model is illustrated in Figs. 2.6 and 2.7 for constant values of m=100, a=1, d=30, and u=0.5. Fig. 2.6 shows the time series of concentration (chemographs) that one would obtain by frequent analysis of water samples taken at a particular river station. To get a picture as in Fig. 2.7, one would have to organize synchronous sampling at multiple stations along the river.

2.6.2 Model implementation

The C++ source code listed in Fig. 2.8 implements a simulation model based on the equations from Sec. 2.6.1. It solves Eqn. 2.2 for a fixed station x and an array of times. Thus, it computes a chemograph as in Fig. 2.6. For compatibility with the **mops** package, the executable reads all input data – including the values of the two parameters u and d – from a text file.

To build an executable from the source code in Fig. 2.8, the GNU C++ compiler can be used (see Kneis, 2012b, for mode info). Assuming that the source code is saved in a file 'adModel.cpp', an appropriate command line for compilation would be:

```
g++ -lstdc++ -o adModel adModel.cpp
```

```
#include<iostream>
#include<fstream>
#include<string>
#include<cmath>
using namespace std;

// Analytical solution of the 1-dimensional convection-dispersion eqn.
double c_xt (const double x, const double t, const double m, const double a,
  const double d, const double u)
{ return(m/a/sqrt(4.*3.1415*d*t) * exp(-pow(x-u*t,2.) / (4.*d*t))); }

// Driver program
int main (int argc, char **argv) {
  //////////// Read file names from command line ////////////////////////////////
  string cmd(argv[0]);
  if (argc != 3) {
    cout << "Usage: " << cmd << " inputfile outputfile" << endl; return(1); }
  string name_ifile(argv[1]);
  string name_ofile(argv[2]);
  //////////// Read parameters from input file //////////////////////////////////
  ifstream ifile(name_ifile.c_str());
  if (!ifile.is_open()) {
    cout << "Input file '" + name_ifile + "' not found." << endl; return(1); }
  double x, m, a, d, u, tmax, dt;
  try {
    string key;
    for (unsigned int i=0; i<7; i++) {
      ifile >> key;
      if (key == "x") {ifile >> x;} else if (key == "m") {ifile >> m;}
      else if (key == "a") {ifile >> a;} else if (key == "d") {ifile >> d;}
      else if (key == "u") {ifile >> u;} else if (key == "dt") {ifile >> dt;}
      else if (key == "tmax") {ifile >> tmax;}
      else {throw("Unexpected parameter in input file.");}
    }
  } catch(...) {
    cout << "Cannot read data from '" + name_ifile + "'." << endl; return(1); }
  ifile.close();
  //////////// Open output /////////////////////////////////////////////////////
  ifile.open(name_ofile.c_str());
  if (ifile.is_open()) {
    ifile.close();
    cout << "Output file '" + name_ofile + "' exists." << endl; return(1); }
  ofstream ofile(name_ofile.c_str());
  if (!ofile.is_open()) {
    cout << "Cannot open file '" + name_ofile + "'." << endl; return(1); }
  //////////// Print solution for the times of interest ///////////////////////
  double t=0.;
  while (t <= tmax) {
    if (t == 0.) { ofile << "t\tc" << endl << t << "\t" << 0. << endl; }
    else { ofile << t << "\t" << c_xt(x,t,m,a,d,u) << endl; }
    t= t + dt;
  }
  //////////// Finish //////////////////////////////////////////////////////////
  ofile.close();
  return(0);
}
```

Figure 2.8: *C++ source code of the advection-dispersion model (file: adModel.cpp).*

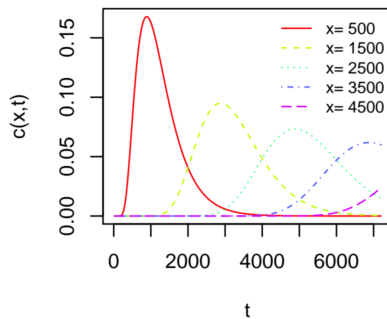


Figure 2.6: *Solutions of Eqn. 2.2 at a fixed set of river stations for variable times.*

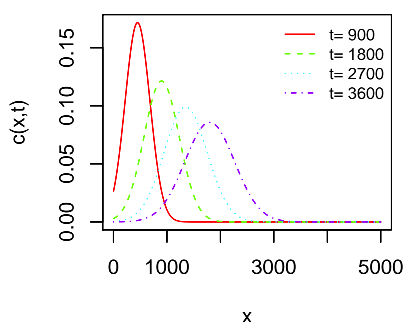


Figure 2.7: *Solutions of Eqn. 2.2 at a fixed set of times for variable river stations.*

```
u 0.5
d 50.
a 15.
m 5.
x 1000.
tmax 7200.
dt   120.
```

Figure 2.9: *Sample input file for the model defined in Fig. 2.8.*

The command should create an executable 'adModel' (possibly with an appropriate file name extension).

To successfully run the model, two command line arguments must be passed to the executable. The first argument is the name of the input file containing all input data, including the values of the parameters u and d. The second command line argument is the name of the output file containing the simulated concentrations. An example of an appropriate input file is given in Fig. 2.9. The model expects to find the definition of a single input value per line. Each line should start with a string, representing the name of the input item, followed by one or more blank characters, followed by the value. The meaning of the values u, d, a, m, and x is clear from Eqns. 2.1 and 2.2. The two additional values define the array of times of interest. $tmax$ defines the upper limit of the simulation time (starting at zero) and dt specifies the temporal resolution of the output. Both values are in units of seconds. The value of dt should be less than $tmax$ to produce (any) useful output.

2.6.3 Observed data

Eqn. 2.2 can be used to determine the average flow velocity u and the longitudinal dispersion coefficient d from tracer experiments. We assume that the set of observations given in Fig. 2.10 was obtained in such an experiment. In addition, we assume that

- The observations were made at river station x= 1000 m while the injection took place at x=0.
- The river's wet cross-section area is a= 15 m^2.
- The known input mass of the tracer was m= 5 g.

```
time    conc
900     0.0001
1200    0.0002
1800    0.00032
3000    0.00015
3600    0.0001
```

Figure 2.10: *Time series of observed concentrations (file: observations.txt).*

```
u {u}
d {d}
a 15.
m 5.
x 1000.
tmax 7200.
dt   120.
```

Figure 2.11: *Template parameter file with placeholders for the values of the parameters u and d (cf. Fig. 2.9).*

2.6.4 Monte-Carlo experiment

Preparation of template file(s)

The first step of setting up a Monte-Carlo simulation is the preparation of parameter template files. Since our model (Fig. 2.8) reads all data from a single file, we have to prepare a single template file only. To do so, we simply take the sample file from Fig. 2.9 and substitute the values of the parameters of interest by appropriate placeholders. Using curly braces as the designated characters for placeholders, an appropriate template file could look as in Fig. 2.11. In this example, the placeholders are named like the parameters. This is not a must but a simple and useful convention.

Writing a driver script in R

The next step is to write an R script to call the `run_mcs` method from the **`mops`** package with appropriate arguments. For our example, such a driver script is shown in Fig. 2.12. The R script has been structured into several parts using comment lines. The subsequent paragraphs provide some details on the contents and meaning of these parts.

PART 0 This part accounts for initial actions. It clears R's memory of variables, loads the **`mops`** package, and sets the working directory. When using a copy of the code, the working directory needs to be adjusted to the local settings.

PART 1 In this part, the names of all files are defined which are created in part 3 of the script. This includes the model's input file (which is generated from a template) and the model's output file. It may be convenient to use temporary file and folder names.

PART 2 In this part, two tables are created (as `data.frame` objects). The first table defines the sampling ranges of the parameters and is passed as argument `ranges_table` to the `mcs_run` method in part 3. The second table holds information required for the automatic editing of the model's input files. It is passed as argument `updating_table` to the `mcs_run` method in part 3. In this example, the table consists of a single row because only a single file needs to be updated.

In the case of models with a larger number of parameters or multiple input files that need to be updated, is is better style to put the contents of the two table(s) in text file(s). Then, one can use R's `read.table` method to read the data and instantiate the `data.frame` objects.

PART 3 This part contains the call to the `run_mcs` method. Information on all arguments can be found in the **`mops`** package's help files. Some additional hints related to the more complex arguments are given below:

- In this example, it is assumed that the file name of the model executable is 'adModel' and that the file resides in the working directory (argument `model_path`). The actual name depends on the output created by compiling the source code from Fig. 2.8.

- The model executable expects two *unnamed* command line arguments (see code in Fig. 2.8). The two expected file names are supplied in an *unnamed* vector to the argument `model_args`.

- In this example, it is assumed that the observed data (Fig. 2.10) reside in a file named 'observations.txt' (argument `obs_file`). Furthermore, it is assumed that this is a TAB-separated file with the time in column 'time' and the observed values in column 'conc'.

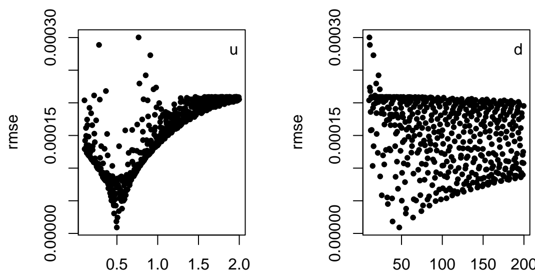


Figure 2.13: *Output graphics created by the R code from Fig. 2.12.*

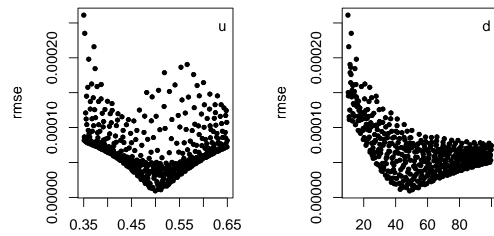


Figure 2.14: *Output graphics created by the R code from Fig. 2.12 after narrowing of the sampling ranges (u in 0.35 ... 0.65; d in 10 ... 100).*

- The functions assigned to the arguments `sim_timeConv` and `obs_timeConv` are identical. They convert a numical time into a value of R's `POSIXct` class. Here, the numerical time is the number of seconds after tracer injection (see Fig. 2.10 and the source code in Fig. 2.8). It does not matter which base time is used (here 1970-01-01 00:00:00) as long as it is identical for the observed and simulated time series.

- The value of the function assigned to argument `gof_function` is returned as a *named* vector. Using a named vector has the advantage that this name appears in the result file.

PART 4 This final part calls `mcs_eval` to analyze and visualize the result of the Monte-Carlo experiment. For each parameter included in the experiment, a the marginal distribution of the model error is plotted in a separate sub-figure (see examples in Figs. 2.13 and 2.14). These plots are also known as 'dotty plots'.

Inspecting the output

The graphical output from the R code (Fig. 2.12) is presented in Fig. 2.13. In this example with only two parameters (u and d), some structure is visible in the plots of the marginal distributions. From the left sub-figure, one may conclude that a parameter value of $u \approx 0.5$ fits best with the observation data. The right sub-figure of Fig. 2.13 suggest that reasonable values for the dispersion coefficient d are probably found in a range between 10 and 70. In order to refine the estimate, one could try different options:

1. Re-run the Monte-Carlo experiment with narrower sampling ranges. For example, the sampling range for parameter d could be restricted to 0 ... 100. This option was already discussed in Sec. 2.4.4. The result of such a re-run with narrowed sampling ranges is presented in Fig. 2.14.

2. Re-run the Monte-Carlo experiment with a higher number of random samples.

3. Use the current estimates of $u = 0.5$ and $d = 50$ as initial values for one of the true optimization algorithms provided by R's `optim` method.

2.7 Using mops with an echse-based model

This section contains some practical hints for the use of **mops** together with an **echse**-based model. The hints refer in particular to the methods `modelError_multiDim` and `modelError_oneDim` which have many arguments in common.

Command line arguments An **echse**-based model generally expects some mandatory command line arguments of the form 'key=value' (see Kneis, 2012a, for details). All optional command line arguments (configuration data) are of the 'key=value' form as well. Therefore, the value assigned to the method's argument `model_args` must always be a *named* vector. A minimum exaple, covering only the mandatory command line arguments is given in Fig. 2.15.

Clean-up functions An **echse**-based model never overwrites existing files. Therefore, when carrying out a series of model runs, there must be a mechanism to

```
############ PART 0 ############################################################
rm(list=ls())
library("mops")

# Set working direcory ( A D J U S T   T O   Y O U R   S E T T I N G S  !)
setwd("~/progress/echse/echse_doc/tex/tools/current/chapters/mops/fig/mcs_example")

############ PART 1 ############################################################
# Define variables for folders and files
dir_run= paste(tempdir(),"run",sep="/")       # Output of current run
dir_mcs= paste(tempdir(),"mcs",sep="/")       # Output of MCS
if (!file.exists(dir_run)) dir.create(dir_run)
if (!file.exists(dir_mcs)) dir.create(dir_mcs)

ifile= paste(dir_run,"input.txt",sep="/")  # Model input (generated from template)
ofile= paste(dir_run,"output.txt",sep="/") # Output file of the model

############ PART 2 ############################################################
# Table with sampling ranges for the parameters
rangeTbl= data.frame(parameter= c("u", "d"), min= c(0.1, 10), max= c(2.0, 200))

# Table defining the files to be edited automatically
updateTbl= data.frame(file_template=c("in_template.txt"), file_result=c(ifile))

############ PART 3 ############################################################
# Call to the Monte-Carlo method from package mops
mcs_run(
  ranges_table= rangeTbl,
  nSamples= 500,
  updating_table= updateTbl,
  model_path= "./adModel",         # on Linux, the "./" is required
  model_args= c(ifile, ofile),
  outdir_model= dir_run,
  outdir_mcs= dir_mcs,
  silent=FALSE
)

############ PART 4 ############################################################
# Analyze the results of all model runs and create useful graphics
mcs_eval(
  outdir_mcs= dir_mcs,
  obs_files= c(all="observations.txt"),
  obs_colsTime= "time",
  obs_colsValue= "conc",
  obs_colsep="\t",
  obs_timeConv= function(x) { ISOdatetime(1970,1,1,0,0,0) + x },
  sim_file= basename(ofile),
  sim_colTime= "t",
  sim_colValue= "c",
  sim_colsep="\t",
  sim_timeConv= function(x) { ISOdatetime(1970,1,1,0,0,0) + x },
  obs_nodata= -9999,
  periods= data.frame(begin=c(), end=c()),
  gof_function= function(obs,sim) { c(rmse= sqrt(mean((sim-obs)^2))) },
  showStats=TRUE
)

print(paste("Outputs, incl. a PDF with graphics, are in '",dir_mcs,"'.",sep=""))
```

Figure 2.12: *R code to run the Monte-Carlo experiment using the* ***mops*** *package.*

```
model_args= c(file_control="myModel.cnf",
  file_log="myModel.log",
  file_err="errors.html",
  format_err="html", silent="true")
```

Figure 2.15: *Example, demonstrating the use of the* `model_args` *argument.*

delete the output created by the previous run. By assigning a proper function to the arguments `func_first` and an appropriate value to `moreArgs_first` such an automatic clean-up may be accomplished. Alternatively, the clean-up may be done right after a model run, by supplying an appropriate function for `func_final` and value for `moreArgs_final`.

In some cases, a complete removal of the model outputs after each run may be undesired. For example, one might be interested in the actual time series output produced by the individual runs of a Monte-Carlo simulation. Then, renaming is usually an appropriate alternative to deletion of files. This can be done, for example, by supplying appropriate code as `func_final` and a suitable value for `moreArgs_final`. In those cases, it is often convenient to use a string representation of the system's time for the generation of unique file names.

Files created by the model Output files created by an **echse**-based model conform to some simple standards which facilitates the use of **mops**. In all time series output files, the column with time information is named 'end_of_interval'. Thus, this is the string to be assigned to the argument `sim_colTime` (Fig. 2.16). Furthermore, the time information printed by **echse**-based models is always in ISO 8601 format (YYYY-MM-DD hh:mm:ss) with date and time separated by a single blank character. This is also the default format used by **mops**. Therefore, the actual argument for `sim_timeConv` can be omitted in the call to methods like `modelError_MCS`. If, even though, a time-conversion function is specified it should be defined as in Fig. 2.16.

```
sim_colTime=  "end_of_interval"
sim_timeConv= function(x) {
  as.POSIXct(strptime(x,
  "%Y-%m-%d %H:%M:%S",tz="GMT"),
  tz="GMT")
  }
```

Figure 2.16: *Appropriate values of the arguments* `sim_colTime` *and* `sim_timeConv` *when using* ***mops****'s methods with an* ***echse****-based model.*

2.8 Troubleshooting

2.8.1 General recommendations

The methods contained in the **mops** package perform quite complex tasks and there is a high potential for making mistakes. In many cases, the cause of failure should be obvious from the generated error messages. However, there are situations where it may be difficult to locate and identify the actual problem. Therefore, some general guidelines for troubleshooting are given in this section. More specific problems are addressed in Sec. 2.8.2.

If one of **mops**'s high-level methods involving a call to a simulation model fails, one should first try to find out at which step of the computation the problem occurred. There are four possible categories a–d with typical symptoms:

(a) Stop occurs prior to running the model

- The model does neither produce the desired output nor any other files (log files, files with error messages, etc.).
- A return code of the model is *not* reported.

(b) Stop occurs when calling the model

- As in case (a), the model does not produce any output files.
- A *non-zero* return code of the model *is* reported.

(c) Stop occurs while running the model

- The model produces a file with error messages and/or a log file whose entries indicate early termination.
- A *non-zero* return code of the model *is* reported.

(d) Stop occurs after running the model

- The model produces the desired output files and possibly a complete log file.

- A return code of the model is *not* reported.

Note that (b) is a quite common case. It typically occurs if the model is called with invalid or incomplete command line arguments. In case of **echse**-based models, it may happen that the model executable is called successfully but the names of the files for log info and diagnostic messages are not properly passed via the command line. Consequently, the model terminates without creating any log or error output. The non-zero return level is the only sign of failure then.

2.8.2 Specific recommendations

The model does not run at all

- Check the path to the executable if a relative or absolute path was specified.
- If the executable is called just by its name but the file does not reside in R's working directory, you might need to check/adjust your 'PATH' environment variable.
- Check for broken links (if a link is used).
- Check whether the file is actually executable.

The command line is not passed to the model

- Try to call the executable directly, i. e. not via an intermediate shell script and not using a link. This is best done by adding the directory containing the executable to the 'PATH' variable.
- If this does not help, replace the executable by a shell script that reports its full command line but does nothing else. Try to learn from this.

The model crashes due to floating point exceptions

In true optimization as well as in Monte-Carlo simulation, the simulation model is run with many different parameter sets. In general, the tested parameter sets are chosen by the used algorithm, based on sophisticated rules or simply by random sampling. Thus, the particular sets are not know a-priori.

In many models, the values of different parameters are not totally independent. For example, if a fictive parameter 'maximumCapacity' has a value of 5, it does not make sense if a companion parameter 'minimumCapacity' has a value greater than 5. Similarly, some parameter values may be incompatible with the initial value(s) of state variable(s). If, for example, the fictive parameter 'minTemperature' has a value of 5, problems are waiting to happen if a related state variable 'temperature' is initialized to a value of 0.

The consequences are as follows:

- When using one of **mops**'s methods together with a true optimization algorithm, it may be necessary to set box-constraints for critical parameters. Similarly, when doing Monte-Carlo simulations with the `modelError_MCS` method, the sampling ranges must be chosen with care in order not to produce invalid (e. g. unphysical) parameter combinations.
- The initial values for the state variables must be chosen so as to be compatible with any parameter sets possibly tested during optimization or Monte-Carlo simulation, respectively.

Disregard of this often causes a model to crash due to floating point exceptions. In other cases, the model only produces implausible outputs (which are hopefully detected because of a bad fit).

Chapter 3

Filling of gaps in meteorological time series (`meteofill`)

3.1 Purpose

For hydrological modeling, time series of meteorological variables are required. Besides rainfall, this usually includes variables like temperature, radiation, humidity, windspeed, and possibly air pressure. The time series of meteorological observations often containg some (or many) missing values due to failure of sensors, power blackout, errors in transmission and recording of data and various other causes.

Internally, all hydrological models require continuous (i. e. gap-free) time series of the meteorological forcing variables. This requirement can be fulfilled in two different ways. In the first strategy (Fig. 3.1, left) the model reads the incomplete series and must fill the gaps internally. The second strategy (Fig. 3.1, right) uses an external pre-processor to complete the information before calling the model.

The second approach has a number of advantages listed below:

- Models are often run many times using the same input data set (for example in calibration or uncertainty analysis). It would be a massive waste of computer time to repeatedly pre-process the time series input in every single run.
- Separating the time series pre-processor from the actual model makes the software more modular which is advantageous in terms of re-use and maintenance.

`meteofill` is designed for the second approach (Fig. 3.1, right). It is a pre-processor to produce gap-free time series from observation data.

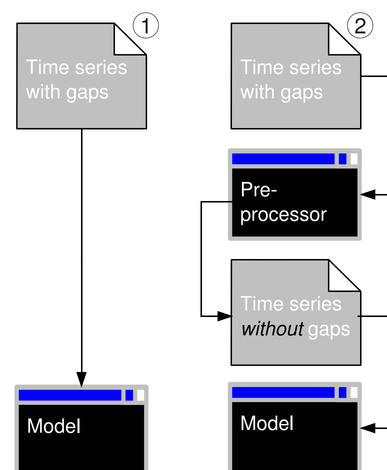


Figure 3.1: *Alternative strategies to handle gaps a model's input time series.*

The methodology used by **meteofill** is built on a space-for-time trade approach (see Sec. 3.2). It requires that observation data are available for *multiple stations*. Using **meteofill** to fill gaps in a time series of observations at a single location is technically possible but it is likely to give poor result (see details in Sec. 3.2.1).

Note that, for *spatially distributed* models, there are now *two separate steps of spatial interpolation*. In the first step, **meteofill** is uses spatial interpolation to remove any gaps in the model's multi-location input time series. In the second step, typically called regionalization, the pre-processed multi-location data are then interpolated to the spatially distributed model objects.

This second interpolation step is typically fully independent of **meteofill** and may be carried out, for example, by the simulation model, or any geostatistical software.

It is also possible, however, to use **meteofill** to combine the two steps of spatial interpolation mentioned above. This is illustrated in Fig. 3.2.

In the left branch of Fig. 3.2, **meteofill** is used to fill the gaps in time series of actual observations made at real-world stations. The distributed model reads the pre-processed time series and performs the regionalization to the model objects internally. In this approach, the model reads a rather small amount of data from files which is efficient with respect to computation time and disk space. User's of **echse**-based models propably always want to use this approach.

The alternative is shown in the right branch of Fig. 3.2, where **meteofill** is used as an all-in-one interpolation tool. This is practically achieved by considering the locations of the model objects as 'normal' observation sites *that never recorded any data*. This kind of usage may be problematic for models with many spatially distributed objects, in particular if longer time series are processed. The produced files may become extremely large, which may result in a massive slowdown of model performance and disk operations in general. Thus, the use of **meteofill** as an all-in-one interpolation tool is recommended in special cases only (small models, few time steps; see example in Fig. 3.6).

3.2 Methods

3.2.1 Filling of gaps

The approach taken by **meteofill** is best explained with an example (Fig. 3.3). In order to substitute a missing data value (cross) by a reasonable estimate, one could, in theory:

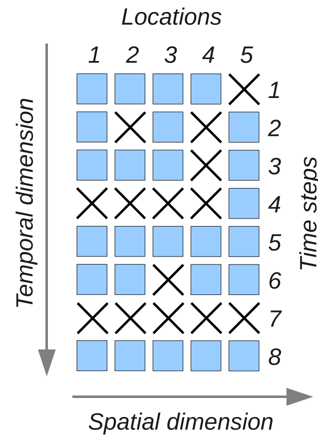


Figure 3.3: *Example of a multi-location times series of a meteorological variable with valid (squares) and missing data values (crosses).*

1. interpolate in time. For example, the missing value at location 3 in time step 4 could be substituted by interpolating between the values at the same location in time steps 3 and 5.

2. assume persistence. For example, the missing value at location 3 in time step 4 could simply be substituted by the value from the previous time step (step 3, same location). This is actually a special case of interpolation in time, where the previous and next value are weighted with factors of 1 and 0, respectively.

3. interpolate in space. For example, the missing values at locations 2 and 4 in time step 2 could be estimated by spatially interpolating the values from the remaining locations 1, 2, and 5 observed at the same time step.

4. combine the ideas of spatial and temporal interpolation.

The current version of **meteofill** primarily relies on *spatial* interpolation, rather than interpolation in time. This can be summarizes by the following simple rules:

- If, in a time step, a value is available at $n \geq 1$ location(s), missing data at the other m locations

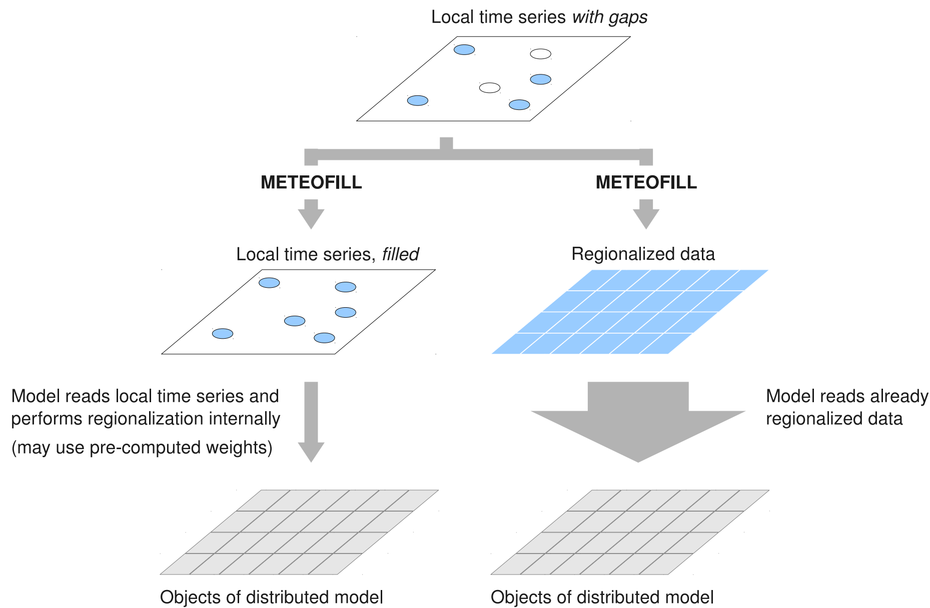


Figure 3.2: *Possible strategies of preparing the time series input of a spatially distributed model using* **meteofill**. *Whenever possible, the left branch should be preffered for performance reasons.*

are estimated by spatial interpolation (see Sections 3.2.2 and 3.2.3 for details). In the simplest case with $n = 1$, the single observation is assumed to be valid globally.

- If, in a time step, no data are available at any location, persistence is assumed. Thus, for each location, the value (or estimate[1]) from the previous time step is used (see Sec. 3.6.1 for practical advices).

After filling the gaps at all station in a time step j, the computation proceeds with time step $j + 1$. This algorithm obviously requires that, in the very first time step, there is at least one location with a non-missing value.

3.2.2 Inverse-distance approach

The inverse-distance method (Eqn. 3.1) is used to perform the spatial interpolation. It is a robust and computationally cheap method. To a limited extent, the spatial autocorrelation of the interpolated variable can be taken into account by adjustment of the exponent p in Eqn. 3.1.

$$y_k = \frac{\sum_{i=1}^{n} (y_i \cdot d(i,k)^{-p})}{\sum_{i=1}^{n} (d(i,k)^{-p})} \tag{3.1}$$

with

y_k	Value at the target location (index k).
y_i	Value at source location with index i.
$d(i,k)$	Distance between locations with indices i and k.
p	Parameter (typically set to 2).

For a particular target location (index k in Eqn. 3.1), the set of appropriate source locations (indices 1 through n in Eqn. 3.1) is found by a sector search. Thus, the surrounding area of the location k is dub-divided into a number of sectors and only the nearest source location is picked from each sector. This is illustrated in Fig. 3.4.

Given a fixed number of sectors, the selection of the source location obviously depends on the orientation of the sectors. To achieve an optimum result, **meteofill** allows for testing different orientations by rotating the sectors (Fig. 3.5). From the tested orientations, **meteofill** uses the one where the cumulated

[1] If data are missing for a number of subsequent time steps

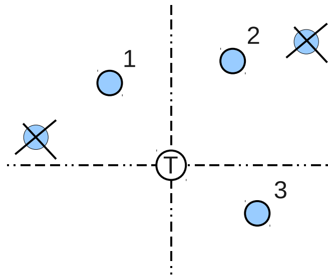


Figure 3.4: *Example of a sector search in spatial interpolation. From each of the four sectors, only the nearest station (1,2,3) is used in estimating the variable at the target location (T). Values at the crossed locations are neglected.*

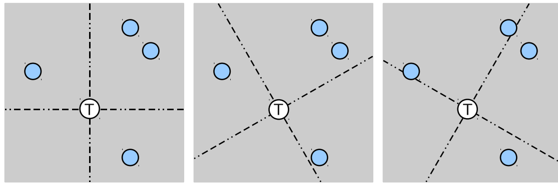


Figure 3.5: *Example of sector rotation. This example used a number of four sectors with 3 different orientations.*

distance between the target and the source locations is minimal.

The selection of the appropriate source locations is the computationally most demanding step. This is especially so if both the number of target and source locations is large. Therefore, **meteofill** carries out a search for the source locations only

- at the very beginning of the computation (first time step).
- if the set of locations with missing data has changed from the previous to the current time step.

3.2.3 Residual interpolation

The result of spatial interpolation can often be improved by using additional predictor variable(s). Values of these valiable(s) must be known at all source and target locations. For example, elevation my be a useful additional predictor when interpolating air pressure or temperature data. Two prominent approaches to spatial interpolation with additional predictors are (1) external drift Kriging and (2) residual interpolation.

meteofill supports the latter approach with the following settings/restrictions:

- Only a *single* predictor variable is supported.
- A *linear* relation between the predictor and the interpolated variable is assumed.
- The *additive* approach is used (not the alternative, multiplicative approach).

The basic algorithm of additive, uni-variate inverse-distance residual interpolation is summarized in Eqns. 3.2 to 3.4

$$y_k = E_k + \frac{\sum_{i=1}^{n} (R_i \cdot d(i,k)^{-p})}{\sum_{i=1}^{n} (d(i,k)^{-p})} \tag{3.2}$$

$$E_j = a \cdot z_j + b \tag{3.3}$$

$$R_i = y_i - E_i \tag{3.4}$$

with the additional symbols (cf. Eqn. 3.1)

E_j Estimate of variable y at an arbitrary location j obtained from a linear model with the external predictor variable z and empirical coefficients a and b.

R_i Residual at a source location with index i.

The coefficients of the linear model (Eqn. 3.3) are updated in every single time step. This is done using the data from all source locations (i. e. all locations with valid data).

The linear correlation between the interpolated variable y and the additional predictor variable z may be more or less strong. In particular, the sign and quality of the correlation may be variable in time. Therefore, **meteofill** accepts a user-specified quality threshold, representing a minimum R^2. If the value of R^2 for the linear model (Eqn. 3.3) is equal or greater than the user-specified threshold, residual interpolation is used (Eqns. 3.2 to 3.4). Otherwise, in the case of a weak correlation, the plain inverse-distance method is applied (Eqn. 3.1). The same is true if the number of data pairs for estimation of the linear model is lower than a user specified minimum sample size.

Residual interpolation is *never* used if the threshold for R^2 is set to a value > 1 or if the minimum sample size is set to a value greater than the total number of locations.

In some cases, use of the linear model may result in undesired extrapolation effects. This is especially so, if

- the range of possible values of the interpolated variable is limited. Example: Precipitation intensity cannot be negative.
- the value of the predictor variable z at a target location is outside the range of z values at the source locations. Example: Data are missing for a location at sea level and all available data correspond to elevations of 500–5000 meters.

In order to suppress such undesired results, **meteofill** allows for data truncation. The result values of residual interpolation are truncated if they are outside a user-specified range.

An example, illustrating the effect of residual interpolation in comparison with plain inverse-distance interpolation is shown in Fig. 3.6.

3.3 Arguments and invocation of meteofill

meteofill is an application written in C++. It expects all input to be supplied as command line arguments. All arguments must be supplied in a keyword-values style as in the following example call:

```
meteofill ifile_locations="locations.txt"
  ifile_data="data_withGaps.txt"
  chars_colsep=" " chars_comment="#"
  nodata=-99. idw_power=2
  nsectors=4 norigins=3
  resid_nmin=3 resid_r2min=0.36
  resid_llim=-40. resid_ulim=40.
  ofile_data="data_filled.txt"
  ofile_locations="locations.txt"
  ndigits_max=1 logfile="log.txt"
  overwrite=true
```

The meaning of the various keywords is as follows:

`ifile_locations` (*string*) Input file listing the spatial coordinates for all locations. See Sec. 3.4.1 for details.

`ifile_data` (*string*) Input file with multi-location time series data. See Sec. 3.4.2 for details.

`chars_colsep` (*character(s)*) One or more character(s) used as a column-separator. Should be quoted, if a special character like TAB (ASCII code 9) is used. Any of the characters (if more than one) is treated as a column separator when reading input files. The columns of output files are separated by the first (or only) character in the set. Typically, a TAB character (enclosed by quotes) is used as in the above example.

`chars_comment` (*character*) Initial character of comment lines in input files (typically the hash character). Should be quoted.

`nodata` (*numeric*) Value to indicate a missing value in the input time series. Typically, large negative values like -99 or -9999 are used for the common meteorological variables.

`idw_power` (*numeric*) Value of parameter p in Eqn. 3.1. Typically a value of 1 or 2. The optimum value may be determined by cross-validation or variogram analysis.

`nsectors` (*integer*) Number of sectors to be used in the search of source locations (see Fig. 3.4). Using `nsectors=1` forces a nearest-neighbor interpolation. You probably want to use a value between 3 and 8 (or 1).

`norigins` (*integer*) The number of sector origins (sector rotations) to be tested. See Sec. 3.2.2 (Fig. 3.5). You probably want to use a value between 1 and 5. Larger values should give better results at the expense of an increase in computation time.

`resid_nmin` (*integer*) Minimum number of locations with valid data (source locations) for possibly activation of residual interpolation. If the actual number of source locations is less than `resid_nmin`, residual interpolation is *not* used. Otherwise, the decision depends on the quality of the linear correlation (see `resid_r2min`). Reasonable values for `resid_nmin` are probably ≥ 3.

`resid_r2min` (*numeric*) Minimum R^2 of the linear model used in residual interpolation (see Sec. 3.2.3). For actual activation of residual interpolation, the computed R^2 for the particular time step must be ge `resid_r2min` and, *in addition*, the sample size must be large enough (see `resid_nmin`). Reasonable values for `resid_r2min` are probably ≥ 0.36 (correlation coefficient of 0.6).

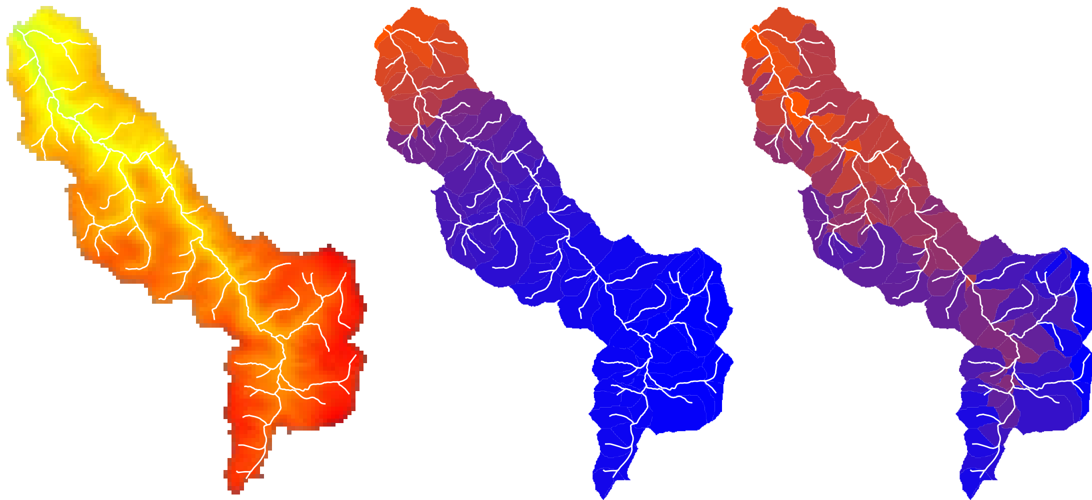


Figure 3.6: *Effect of residual interpolation on the regionalization of air temperature in a mountainous watershed. In this example, the air temperature observed at climate stations is highly correlated to elevation. The stations (not shown) are located outside the watershed in the North (low elevation, warm) and South-East (high elevation, cold). Left: Elevation model (red: high, green: low) and river net (white). Center:* **meteofill** *output using the plain inverse-distance approach (blue: cold, red: warm). Right:* **meteofill** *output using residual interpolation with elevation as additional predictor. Obviously, higher temperatures are predicted in the valleys by taking the vertical temperature gradient into account.*

`resid_llim` (*numeric*) Lower truncation limit in case of residual interpolation (see Sec. 3.2.3). For variables that cannot take negative values (like precipitation or short-wave radiation), `resid_llim` should be set to zero.

`resid_ulim` (*numeric*) Upper truncation limit in case of residual interpolation (see Sec. 3.2.3).

`ofile_data` (*string*) Name/path of the output file containing the time series with all gaps filled according to the method described in Sec. 3.2.

`ofile_locations` (*string*) Name/path of the output file containing a list of the locations from `ifile_locations` for which data are actually present in `ifile_data`.

`logfile` (*string*) Name/path of an output log file.

`ndigits_max` (*integer*) Number of digits to be used in the output file `ofile_data`.

`overwrite` (*logical*) A value of either true of false. If true, any existing output files will silently be replaced.

After successful execution, the return code of **meteofill** is zero. If the program terminates due to an error, a non-zero code is returned and traceback info is sent to standard output.

3.4 Input

3.4.1 Locations table

The locations table is a text file with four columns (see Sec. 3.3 for how to select a column separator). The expected column names are `id`, `x`, `y`, and `z`. The `id` column contains the names of locations (usually climate stations or rain gages), for which data are available. The IDs are read as strings. The corresponding spatial coordinates go in the `x` and `y` fields. For the coordinates one should use a *geodetic* system (i. e. units of meters, kilometers, miles, etc.). The `z` field should contain the values of the external predictor variable for residual interpolation. Elevation is the natural choice if nothing better is available. If residual interpolation should not be used anyway, the `z` column may be filled with dummy values.

An example of a locations table is given in Fig. 3.7.

It is OK if the locations table contains more than the mandatory columns. **meteofill** simply ignores the unnecessary information. It is also OK if the locations table contains records for additional locations not

```
id        x          y       z
klotzsche            4623183 5667440 227
hosterwitz           4629789 5655359 114
strehlen             4623996 5652991 119
dipps_rein           4620226 5644000 365
zinnwald             4623539 5622933 877
```

Figure 3.7: *Example of a locations table.*

```
time     MT.CMP. ARIES   MT.ORO
2000-01-01       0       -99     0
2000-01-02       4       -99     1
2000-01-03       12      3       1
2000-01-04       -99     -99     -99
2000-01-05       0       -99     0
```

Figure 3.8: *Example of multi-location time series file for use with* **`meteofill`**.

present in the time series input file (Sec. 3.4.2). These records are silently ignored as well.

3.4.2 Time series file

The time series input file is a text file with $n+1$ columns where n is the number of locations (see Sec. 3.3 for how to select a column separator). The table must have a header line with column names. The first column of the file is expected to contain time information in ascending order (oldest record first). Any format can be used as **`meteofill`** internally treats the data as strings, not as times. For this first column, an arbitrary name can be chosen (but it must be present). The remaining n columns contain the numeric observation data at the n locations. These columns may be in any order but their names need to match exactly with the location IDs provided in the locations table (Sec. 3.4.1). Any missing or invalid data values must be indicated by the `nodata` value (see Sec. 3.3). An example of multi-location time series file is given in Fig. 3.8.

For the algorithm to be successful, it is important that the very first (oldest) record contains valid data for one location, at least.

3.5 Output

The current version of **`meteofill`** generates three output files whose names are specified at the command line (see Sec. 3.3). The contents of the files is described in Table 3.1.

3.6 Hints for practical usage

3.6.1 Missing-only data (options beyond persistence)

As pointed out in Sec. 3.2.1, **`meteofill`** assumes persistence if, in a particular time step, no data are available for any station (let's call this situation a *global gap*). Although persistence is a simple and intuitive approach, it is likely to give satisfactory results only as long as the period of missing-only data is short enough. For long-lasting global gaps, persistence may not be a suitable assumption and it may be desireable to fill the gap with predefined values. Possible candidates are monthly average values or, in the case of rainfall data, a fixed value of zero.

Furtunately, there is a simple trick to let **`meteofill`** fill such global gaps with predefined values. All you need to do is to supplement the input data with a *synthetic* station that 'recorded' the desired fill-in values (which may vary in time). The spatial coordinates of that synthetic station must be chosen so that it is *very far* away from all the actual stations (e. g. some million kilometers). Hereby it is ensured that the data from the synthetic station will practically be ignored (i. e. weighted with almost zero), as long as any of the actual stations provides valid data.

Table 3.1: *Output files of **meteofill**. The entries in the 'File' column refer to the command line arguments describe in Sec. 3.3.*

File	Contents
ofile_data	A multi-location time series table. Format and contents are identical to the input time series Sec. 3.4.2 except that all nodata values are replaced by estimates according to Sec. 3.2.
ofile_locations	Similar to the input locations table described in Sec. 3.4.1. The table contains only the mandatory columns and list only those stations for which time series data are actually present.
logfile	Lists, for each time step, the number of locations with valid data (as absolute number in column numdata and as a percentage in column availability. The column config_changed contains information on whether the set of locations with missing data has changed in comparison to the previous time step (requiring a new search for neighbored locations). The columns residual_interp and r2 contain information on the use of residual interpolation (true/false) and the corresponding R^2 of the linear model.

Chapter 4

River cross-section analysis (`xsAnalyzer`)

4.1 Purpose

The **`xsAnalyzer`** software computes basic hydraulic properties of a river reach being described by

1. a representative cross-section geometry,
2. the hydraulic roughness (energy loss parameter),
3. the bed slope,
4. the reach length.

The computed results refer to a situation of steady uniform flow. This means that (1) the flow rate is constant in time, (2) the cross-section geometry does not change along the reach, and (3) the slope of the water surface is identical to the bed slope (no backwater).

The **`xsAnalyzer`** tool can be used, for example, to

- calculate an approximate rating curves for ungaged sites at a river,
- estimate parameters to describe a reach's retention characteristics for use by hydrologic flood routing methods.

Note that the R-package **`topocatch`** (see Chap. 1) contains methods to perform similar tasks. However, as opposed to **`xsAnalyzer`** these methods are not capable of handling compound cross-sections, i. e. those with a variable roughness (see Sec. 4.2).

4.2 Methods

The most important input information of **`xsAnalyzer`** is the cross-section geometry. Fig. 4.1 shows a typical example with a distinction between the main channel and the flood plains on either side of the river.

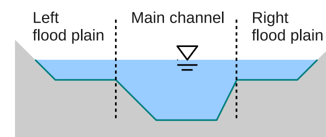


Figure 4.1: *A typical river cross-section.*

The cross-sections's geometry can be represented as a two column table. One column specifies the offset from a fixed point (typically at the left bank) and the other column holds the corresponding elevations. From such a table, the functions $A(D)$ and $R(D)$ can instantly be computed, where D is the maximum flow depth in the cross-section, A is cross-sections the wet area, and R is the hydraulic radius (wet area divided by the wet perimeter).

Using Manning's equation (Eqn. 4.1), the flow rate for steady uniform conditions Q can be calculated for a given flow depth D if the slope S and the roughness parameter n are known. The reverse computation aimed at finding the value of D for a given Q is also possible but requires a numerical solution.

$$Q(D) = \frac{1}{n} \cdot \sqrt{S} \cdot A(D) \cdot R(D)^{2/3} \tag{4.1}$$

For many real-world cross-sections, the use of a single, unique roughness parameter n is not appropriate. This is often the case for cross-sections with flood plains (Fig. 4.1) because the actual surface roughness is horizontally variable. For example, the flood plains

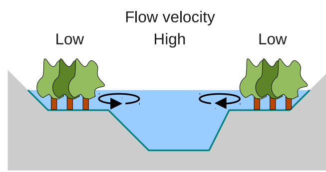


Figure 4.2: *Turbulence due to shear at the interface of flow zones.*

may be covered by vegetation while the bed of the main channel is made of sand or concrete. Then, energy losses due to turbulence will be quite different in the channel and the flood plains.

To handle this situation, **xsAnalyzer** uses the idea of compound cross-sections (see Cunge et al., 1980). Thus, the cross-section is horizontally sub-divided into separate zones. To each zone, a unique value of n is applied when using Eqn. 4.1. In the case shown in Fig. 4.1, for example, the flow rate would be computed in two steps. First, the individual flow rates of the main channel and the two flood plains would be calculated, before the three values are added up in a second step.

It is important to keep in mind that this approach allows for different flow velocities in the zones. In reality, this would lead to additional turbulence and, consequently, result in additional energy losses due to shear at the zones' interface (Fig. 4.2). Such losses, however, are difficult to estimate and they are *totally neglected* by **xsAnalyzer**.

If **xsAnalyzer** detects a water surface elevation which is higher than the cross-section's elevation at the first and/or last offset, it assumes a vertical wall at the respective offset. The output file (Sec. 4.5) contains a field to detect those critical situations.

4.3 Arguments and invocation of xsAnalyzer

xsAnalyzer is an application written in C++. It expects all input to be supplied as command line arguments. All arguments must be supplied in a keyword-values style as in the following example call:

```
xsAnalyzer file_xsection="geometry.txt"
  file_flows="flowsOfInterest.txt"
  file_out="output.txt"
  read_3d=false
  read_roughness=false
  default_roughness=30
  slope=0.000485
  plain_length=1000.
  max_nreserv_kalmil=100
  print_only_routing=false
```

The meaning of the various keywords is as follows:

`file_xsection` (*string*) Input file with cross-section geometry data. See Sec. 4.4.1 for details.

`file_flows` (*string*) Input file listing flow rates of interest. See Sec. 4.4.2 for details.

`file_out` (*string*) File name/path for output.

`read_3d` (*logical*) Must be FALSE if the geometry data are given as a table of corresponding offsets and elevations. Must be TRUE if the geometry data are available as 3-dimensional coordinates.

`read_roughness` (*logical*) Must be TRUE if the table with geometry data contains an additional column with roughness values. Note that the roughness must be specified as Strickler's Kst parameter which is the inverse of Manning's n ($Kst = 1/n$). If FALSE, a unique roughness value is assumed (see next argument).

`default_roughness` (*numeric*) The unique value of the roughness parameter to be used if no values are specified in the geometry file. Note that the roughness must be specified as Strickler's Kst parameter which is the inverse of Manning's n ($Kst = 1/n$).

`slope` (*numeric*) Slope of the river bed as a dimensionless number (i. e. meters elevation per meters distance).

`plain_length` (*numeric*) Length of the reach as a horizontal distance. For typical values of the slope, this is almost identical to the actual length of the reach.

`max_nreserv_kalmil` (*integer*) A parameter to control the output. It specifies the maximum number of conceptual linear reservoirs for the Kalinin-Miljukov routing method.

`print_only_routing` (*logical*) A switch to optionally reduce the amount of output information.

After successful execution, the return code of **xsAnalyzer** is zero. If the program terminates due to an error, a non-zero code is returned and traceback info is sent to standard output.

```
offset	z
-100	20
0	20
5	15
25	15
30	20
100	20
```

Figure 4.3: Example of geometry file in 2D format without roughness information.

```
offset	z	Kst
-100	20	20
0	20	30
5	15	30
25	15	30
30	20	20
100	20	20
```

Figure 4.4: Example of geometry file in 2D format with specified roughness.

```
x	y	z
-100	0	20
0	0	20
5	0	15
25	0	15
30	0	20
100	0	20
```

Figure 4.5: Example of geometry file in 3D format without roughness information.

```
x	y	z	Kst
-100	0	20	20
0	0	20	30
5	0	15	30
25	0	15	30
30	0	20	20
100	0	20	20
```

Figure 4.6: Example of geometry file in 3D format with specified roughness.

4.4 Input

4.4.1 Geometry data

The file holding the geometry data must be in tabular format with columns separated by the TAB character (ASCII character code 9). Lines with an initial # character are treated as comment lines. A header line with column names is mandatory. See the examples below for the expected column names.

Figs. 4.3 – 4.6 illustrate the four different possible file formats for geometry data. If the data are given in 2-dimensional format, the offsets must be in increasing order.

Note that, of roughness information is present, the value for a particular offset applies to the part of the cross-section between this offset and the following offset. Thus, the roughness information of the very last record is effectively ignored (but a value must be present).

4.4.2 Flow values of interest

The flow data are read from a plain text file like the one in Fig. 4.7. This is effectively a single-column table. A header must not be present. The user is free to specify a list of flow values with arbitrary increments but the data must be in increasing order.

4.4.3 Units

Note that the units of all input data need to be consistent. If, for example, the offsets are given in meters, the elevation data must be provided in units of meters too. At the same time, flow rates must be given in units of m^3 per second.

```
0.1
1
2
5
10
50
100
```

*Figure 4.7: Example of file with flow data for use with **`xsAnalyzer`**.*

```
# Cross-section property table
#   X-section data file:    in/montalban/geometry_constRough.txt
#   Flow data file:         in/montalban/flowsOfInterest.txt
#   Used reach length:      1000
#   Used slope:             0.000485
#   Used roughness: 30 (Global default)
flow stage overtop wet_area top_width wet_perimeter volume_total dvdq_total nsub volume_sub dvdq_sub
0.5  19.74 0       1.71     5.6       5.79          1707.2       2231.8     3    569.1      743.9
1    19.92 0       2.82     6.94      7.19          2823.1       2111.8     2    1411.6     1055.9
2    20.16 0       4.69     8.74      9.07          4694.9       1915.7     1    4694.9     1915.7
5    20.64 0       10.84    18.02     18.57         10836.9      1932.4     1    10836.9    1932.4
10   21.02 0       19.54    27.94     28.67         19541.2      1553.4     1    19541.2    1553.4
20   21.41 0       31.33    32.13     32.98         31326.8      1175.3     1    31326.8    1175.3
30   21.75 0       43.05    38.84     39.73         43046.2      1310.8     1    43046.2    1310.8
50   22.33 0       74.82    72.4      73.55         74819.0      1302.6     1    74819.0    1302.6
75   22.64 0       98.45    78.1      79.51         98446.4      849.8      1    98446.4    849.8
100  22.88 0       117.31   78.32     80.04         117309.9     727.7      1    117309.9   727.7
```

Figure 4.8: *Example of an output file produced by* **`xsAnalyzer`**.

4.5 Output

The current version of **xsAnalyzer** generates a single output file containing a summary of the properties of the cross-section / reach. An example is shown in Fig. 4.8.

The meaning of the columns is as follows:

`flow` Flow rates of interest as specified in the input file.

`stage` Water surface elevations corresponding to the flow rates of interest.

`overtop` Either 0 or 1. A value of 1 means that the water surface elevation is higher than the cross-sections elevation at the first and/or last offset. In that case, a vertical wall was assumed as a boundary.

`wet_area` Wet cross-section area.

`top_width` Cross-section top width.

`wet_perimeter` Wetted perimeter.

`volume_total` Total storage volume of the reach.

`dvdq_total` Estimated derivative of the storage volume with respect to the flow rate.

`nsub` Suggested number of linear reservoirs (= length of cascade) for Kalinin-Miljukov routing.

`volume_sub` Storage volume in the individual reservoirs for Kalinin-Miljukov routing.

`dvdq_sub` Estimated derivative of the storage volume with respect to the flow rate for the individual reservoirs for Kalinin-Miljukov routing.

Chapter 5

Time series visualization tool (tsplot)

5.1 Purpose

tsplot is a lightweight tool for the interactive visualization of time series data. It offers a convenient way to quickly inspect the output of an **echse**-based simulation model. It is particularly useful for showing simulated and observed data in a single plot for the purpose of comparison (see Fig. 5.1). Note that **tsplot** is optimized for interactive plotting only. If you need to create plots for presentation in a paper, for example, you better use R commands or another software.

5.2 Required software

tsplot is implemented as an R script. Thus, to use it, a current version of R must be installed. See Kneis (2012b) for information on how to install R. In order to invoke **tsplot** via a bash shell script under Windows, MSYS is required (see Kneis, 2012b).

5.3 Instructions files

The source(s) of the data to be plotted with **tsplot** and the corresponding styles are defined in instructions files. For every individual plot, one needs to create such a file. At a first glance, this may seem inconvenient. However, putting the instructions into a file allows for re-drawing a particular plot with only a single mouse click. In modeling studies – and in particular during model calibration, when the data change frequently – this is of great advantage.

Instruction files are plain text files containing data in a tabular format. The columns must be separated by white spaces (one or more spaces of tab characters). Each line of the file (row in the table) defines a time series to be plotted.

The instructions file must contain a table header with a complete set of column names (in any order). The meaning of the columns is as follows:

`file` (*string*) Name of a text file containing time series data in the format described in Sec. 5.4. Absolute or relative paths can be used. On Windows systems, it is important that the forward slash (/) is used to separate the names of directories, instead of the usual backslash (\).

`colname` (*string*) Name of a column existing in the data file specified in `file`. The values in this column are taken as the y-values. If the data file has no header, i. e. no column names, one has to supply the index of the column rather than a name. The smallest useful index is 2 (see Sec. 5.4).

`color` (*string*) Name of an existing color in R (examples: *red* or *lightblue*). Call the function `colors()` from an R prompt to display a (lengthy) list of all pre-defined color names.

`type` (*string*) This defines the style used for plotting. Using 'p' (for points) and 'l' (for lines) is appropriate for instantaneous data. For regular time series, where the values represent averages (or sums) over a intervals of time, you should 'S' or 's', depending on whether the given times specify the begin ('s') or the end ('S') of an interval. The output time series produced by **echse** models use the latter convention, thus you should use 'S'. Note that using 'p' may slow down the creation of plots significantly if the time series contain many values. In such cases, use one of the other options as an alternative.

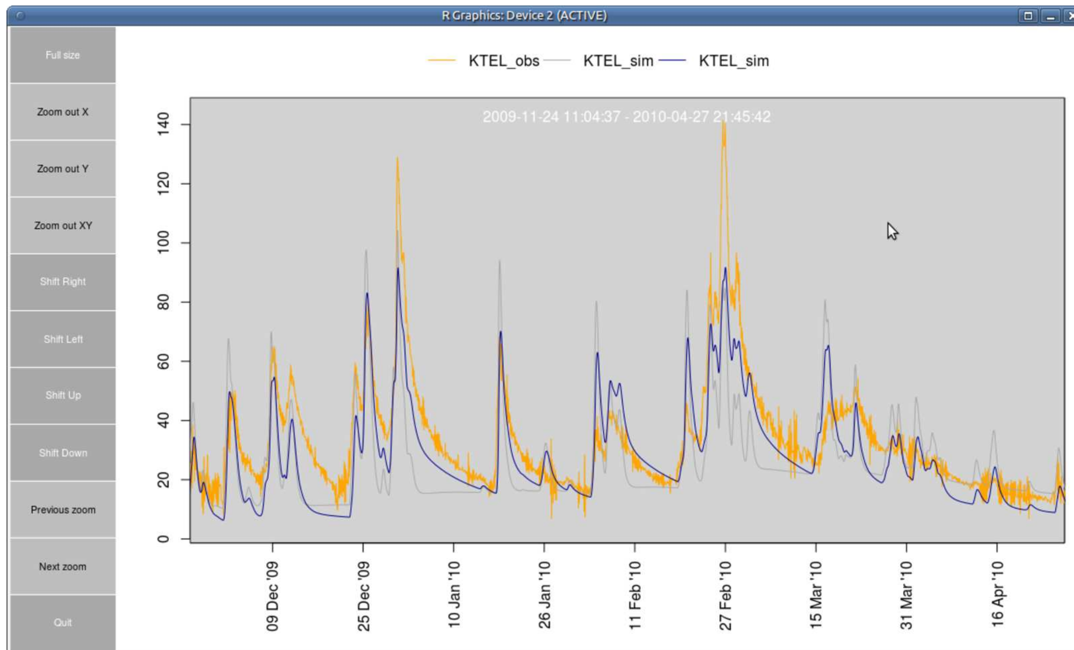


Figure 5.1: *Screen shot of a* **`tsplot`** *application.*

`name` (*string*) A name for the series to appear in the legend.

`header` (*logical*) Must be a logical value (TRUE or FALSE in uppercase letters). If TRUE, **`tsplot`** expects the data file to contain a header with column names.

`comment` (*string*) A single character to be treated as a comment character in the data file. Quite often, the hash character (#) is used for this purpose.

`nodata` (*string*) The value used in the data file to indicate missing values. Typical examples include 'NA' or '-9999'. The corresponding rows of the data file are ignored when creating the plot.

`factor` (*numeric*) A factor to be applied to the data before plotting. A value other that 1 may be useful to re-scale data when plotting multiple series whose values are of different magnitude.

`xcut` (*logical*) Must be a logical value (TRUE or FALSE in uppercase letters). If TRUE, the x-axis is truncated to the range of times contained in the data file. When plotting multiple series, a value of TRUE should not appear in more than 1 row of the instructions file.

An example of an instructions file is given in Fig. 5.2. If you need to put comment lines in the instructions file, use a semicolon as the first character of the line. The line will then be ignored by **`tsplot`** (see example in Fig. 5.2).

5.4 Expected format of data files

The actual time series data must be stored in plain text files. These file(s) must be formatted as follows:

- A tabular format is expected with columns separated by the tab character (ASCII code 9).
- Time information must be in the *first* column of the file.
- Times must be encoded as stings in ISO 8601 format (YYYY-MM-DD hh:mm:ss) with date and time separated by a single blank. Alternatively, one can provide only the date using the format YYYY-MM-DD. If only the date is given, a time of 00:00:00 is implicitly assumed.
- The times in column 1 may be given in regular *or* irregular intervals. They should be in increasing order.

```
file     colname color type  name      header  comment  nodata  factor  xcut

; Example with two series

obs.txt Q       cyan  S     Observed  TRUE    #        NA      1       FALSE
sim.txt qx_avg  blue  S     Model     TRUE    #        NA      1       TRUE
```

Figure 5.2: *Example of a* **tsplot** *instructions file to display two time series of observed and simulated values.*

```
date_and_time_UTC        stat1    stat2
1996-07-26 00:00:00      0.8      1
1996-07-26 01:00:00      11.5     14
1996-07-26 02:00:00      10.1     13
1996-07-26 03:00:00      3.2      3
1996-07-26 04:00:00      5.5      3
1996-07-26 05:00:00      4.7      2
1996-07-26 06:00:00      4.2      4
1996-07-26 07:00:00      3.5      4
1996-07-26 08:00:00      7.8      10
1996-07-26 09:00:00      2.8      3
1996-07-26 10:00:00      3        3
1996-07-26 11:00:00      7.3      9
1996-07-26 12:00:00      0        0
1996-07-26 13:00:00      0        0
```

Figure 5.3: *Example of a time series file for use with* **tsplot**.

- The file can optionally have a header line specifying column names. Such names should be valid names in R, i. e. the first character must be a letter. More letters, digits, and/or underscores may follow.

An example of a properly formatted time series file is shown in Fig. 5.3.

5.5 Invoking tsplot

5.5.1 On Linux

On Linux, one needs to execute the shell script `tsplot.sh` and supply the name of the instructions file as a command line argument.

Using the command line

Assuming that an instructions file `myplot.txt` exists in your home directory, a call from the command line might look as follows:

```
./tsplot.sh /home/myname/myplot.txt
```

To use this, however, one first needs to navigate to the directory of **tsplot**, where the `tsplot.sh` resides. This is rather inconvenient. To be able to call **tsplot** from *any* directory, you could do one of the following:

- Move all files related to **tsplot** to a directory listed in the 'PATH' environment variable.
- Alternatively, add the path of the directory with the **tsplot** sources to the `PATH` environment variable. See Kneis (2012b) for details.
- Alternatively, put a script in one of the directories already contained in the `PATH` variable and let this script call `tsplot.sh`. In that case, you need to make sure that command line arguments are passed on.

Using the file browser's context menu

In Ubuntu, Nautilus Actions provide the most convenient way to process an instructions file with **tsplot**. After defining a new action, you can simply process an instructions file from the Nautilus file browser's context menu. Thus, only two clicks with the right and left mouse buttons are necessary. See the documentation of Nautilus Actions for more info.

5.5.2 On Windows

On Windows, one needs to execute the batch file `tsplot.bat` and supply the name of the instructions file as a command line argument.

Assuming that an instructions file `c:\temp\myplot.txt` exists, a call from the command line might look as follows:

```
tsplot.bat c:\temp\myplot.txt
```

To use this, however, one first needs to navigate to the directory of **`tsplot`**, where the `tsplot.bat` resides. To be able to call **`tsplot`** from *any* directory, you could do one of the following:

- Move all files related to **`tsplot`** to a directory listed in the `path` environment variable.
- Alternatively, add the path of the directory with the **`tsplot`** sources to the `path` environment variable. See Kneis (2012b) for details.

Using the file browser's context menu

There are basically two ways:

- Invent a new file extension for your instruction files (example: '.iii'). Create a new custom file type with that extension. Then configure the default 'open' action for the file type so that `tsplot.bat` is called with the file name as an argument. Thus, the 'open' action should read like `some-path\tsplot.bat "%1"`.
- Alternatively, one can modify the 'open' action of any file type by editing the Windows registry. This is recommended for experienced users only and will not be described here.

List of Figures

1.1 The four spatial input data sets of **topocatch**. 8
1.2 Example of a file in ASCII grid format. 9
1.3 Flow directions encoded as integer values. 10
1.4 Steps of the vector-to-raster conversion. 11
1.5 The approach used to iteratively build the catchments. 11
1.6 Definition of basic properties of a river cross-section. 13
1.7 Selection of parent cross-sections for interpolation. 14
1.8 Interpolation of the cross-section's characteristic functions. 14
1.10 Relation between the flow rate Q and the storage volume V for a linear reservoir and a river reach with an irregularly shaped cross-section. 14
1.9 Example of an output file created by `xs.reachPars`. 15
1.11 Proper and improper junctions in a shape file. 18
1.12 Typical example of a critically short reach in a dense river network. 18
1.13 Representation of a reservoir with two inflows in a map and in the input shape file. 19

2.1 Basic outline of an optimization method. 22
2.2 Basic outline of Monte-Carlo simulation. 22
2.3 Goodness-of-fit for a dynamic simulation model (left) and an empirical linear model (right). 23
2.4 Example application of the `update_template` method. 24
2.5 Another example application of the `update_template` method. 24
2.8 C++ source code of the advection-dispersion model (file: adModel.cpp). 26
2.6 Solutions of Eqn. 2.2 at a fixed set of river stations for variable times. 27
2.7 Solutions of Eqn. 2.2 at a fixed set of times for variable river stations. 27
2.9 Sample input file for the model defined in Fig. 2.8. 27
2.10 Time series of observed concentrations (file: observations.txt). 28
2.11 Template parameter file with placeholders for the values of the parameters u and d (cf. Fig. 2.9). . . . 28
2.13 Output graphics created by the R code from Fig. 2.12. 29
2.14 Output graphics created by the R code from Fig. 2.12 after narrowing of the sampling ranges (u in 0.35 . . . 0.65; d in 10 . . . 100). 29
2.12 R code to run the Monte-Carlo experiment using the **mops** package. 30
2.15 Example, demonstrating the use of the `model_args` argument. 31
2.16 Appropriate values of the arguments `sim_colTime` and `sim_timeConv` when using **mops**'s methods with an **echse**-based model. 31

3.1 Alternative strategies to handle gaps a model's input time series. 33
3.3 Example of a multi-location times series of a meteorological variable with valid (squares) and missing data values (crosses). 34

3.2 Possible strategies of preparing the time series input of a spatially distributed model using **meteofill** . . . 35
3.4 Example of a sector search in spatial interpolation. . . . 36
3.5 Example of sector rotation. . . . 36
3.6 Example to illustrate the effect of residual interpolation. . . . 38
3.7 Example of a locations table. . . . 39
3.8 Example of multi-location time series file for use with **meteofill**. . . . 39

4.1 A typical river cross-section. . . . 41
4.2 Turbulence due to shear at the interface of flow zones. . . . 42
4.3 Example of geometry file in 2D format without roughness information. . . . 43
4.4 Example of geometry file in 2D format with specified roughness. . . . 43
4.5 Example of geometry file in 3D format without roughness information. . . . 43
4.6 Example of geometry file in 3D format with specified roughness. . . . 43
4.7 Example of file with flow data for use with **xsAnalyzer**. . . . 43
4.8 Example of an output file produced by **xsAnalyzer**. . . . 44

5.1 Screen shot of a **tsplot** application. . . . 48
5.2 Example of a **tsplot** instructions file. . . . 49
5.3 Example of a time series file for use with **tsplot**. . . . 49

List of Tables

2.1 Interior of an objective function in the context of dynamic modeling. 23

3.1 Output files of **`meteofill`**. 40

Bibliography

Cunge, J., Holly, F.J., Verwey, A., 1980. Practical aspects of computational river hydraulics. Pitman publishing.

Kneis, D., 2012a. Eco-Hydrological Simulation Environment (ECHSE) - Documentation of the Generic Components. University of Potsdam, Institute of Earth- and Environmental Sciences. URL: http://echse.bitbucket.org/downloads/documentation/echse_core_doc.pdf.

Kneis, D., 2012b. Eco-Hydrological Simulation Environment (ECHSE) - Installation and Administration Guide. University of Potsdam, Institute of Earth- and Environmental Sciences. URL: http://echse.bitbucket.org/downloads/documentation/echse_install_doc.pdf.

Kneis, D., 2013. R-package topocatch: Pre-processing of spatial data for hydrological catchment modeling.

Kneis, D., Bürger, G., Bronstert, A., 2012. Evaluation of medium-range runoff forecasts for a 50 km2 watershed. Journal of Hydrology 414-415, 341–353. doi:10.1016/j.jhydrol.2011.11.005.