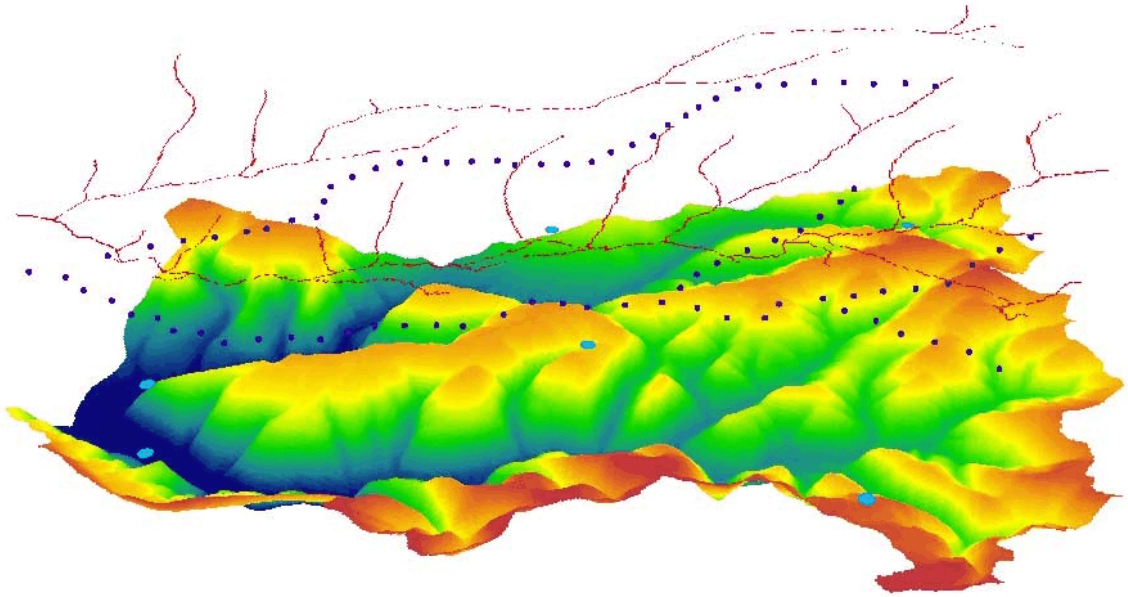
Institut für Hydrologie

der Albert-Ludwigs-Universität Freiburg i. Br.

Uwe Hagenlocher

Das TRAIN-ZIN-Modell – Weiterentwicklung und Anwendung im Dragonja-Einzugsgebiet (Slowenien)



Diplomarbeit unter der Leitung von Prof. Dr. Chr. Leibundgut

Freiburg i. Br., März 2008

Institut für Hydrologie

der Albert-Ludwigs-Universität Freiburg i. Br.

Uwe Hagenlocher

Das TRAIN-ZIN-Modell – Weiterentwicklung und Anwendung im Dragonja-Einzugsgebiet (Slowenien)

Referent: Prof. Dr. Ch. Leibundgut

Korreferent: Dr. Jens Lange

Diplomarbeit unter der Leitung von Prof. Dr. Chr. Leibundgut

Freiburg i. Br., März 2008

Danksagung

Zunächst möchte ich Herrn Prof. Dr. Leibundgut für seine Vorlesungen, Exkursionen und Seminare im Rahmen meines Studiums sowie für die Themenvergabe danken. Herrn Dr. Jens Lange danke ich ebenso für seine Veranstaltungen sowie für die Betreuung dieser Diplomarbeit.

Anne Gunkel danke ich für die wertvolle Hilfe beim Einstieg in die Konzepte von TRAIN und ZIN und den komplexen C++-Code des Modells.

Die Kollegen aus der Rheinstraße 10 waren immer für eine hilfreiche fachliche Diskussion bereit und sorgten nicht zuletzt auch für eine angenehme Arbeitsatmosphäre. Danke Sameer, Dominik, Matthias (auch fürs Korrekturlesen) und Irene!

Nicht zu vergessen sind auch die fleißigen Menschen, die kostenfreie Software anbieten und so jenseits von Profitstreben ihr Können anderen zur Verfügung stellen. Dafür mein herzlicher Dank an die Entwickler von Subversion, TortoiseSVN, WinMerge, VisualLeakDetector und VSE-debug!

Ohne die Unterstützung meiner Eltern wäre mir das Studium nicht möglich gewesen. Für ihre dauernde Hilfsbereitschaft möchte ich mich bedanken.

Für ihre Geduld und liebevolle Unterstützung danke ich Lisa.

Inhalt

I Verzeichnis der Abbildungen VIII

II Verzeichnis der Tabellen X

Zusammenfassung XI

Summary XIII

1 Einleitung 1

2 Zielsetzung und Vorgehensweise 3

2.1 Aufbau der Arbeit 3

3 Werkzeuge der Programmierung 5

3.1 Die Programmiersprache C++ 5

3.1.1 C-Arrays und C++ Container-Klassen 5

3.1.2 Laufzeit-Tests 6

3.1.3 Memory-Leaks und Destruktoren 7

3.1.4 Mehrfach-Verwendung von Code 9

3.2 Die Entwicklungsumgebung Microsoft Visual Studio 10

3.2.1 Solutions und Projekte 10

3.2.2 Debugging 11

3.3 Zusammenarbeit mehrerer Entwickler 11

3.4 Fazit 12

4 Beschreibung des TRAIN-ZIN-Modells 15

4.1 Das ZIN-Modell 15

4.1.1 Niederschlag 15

4.1.2 Verdunstung 16

4.1.3 Abflussbildung 16

4.1.4 Abflusskonzentration 18

4.1.5 Channel-Routing 19

4.1.5.1 Beschreibung des Muskingum-Verfahrens 19

4.1.5.2 Beschreibung des Muskingum-Cunge-Verfahrens 20

4.1.5.3 Beschränkungen des Muskingum-Cunge-Verfahrens 21

4.2 Das TRAIN-Modell 21

4.3 TRAIN-ZIN – Programmstruktur 22

4.4 Fazit 24

5 Modellentwicklung 27

5.1 Das Benutzer-Interface 27

5.1.1 Anforderungen 27

5.1.2 Das ASCII-Interface 28

5.1.3 Die Programm-Seite des Interface 29

5.1.4 Beschränkungen 32

5.2 Verarbeitung von Stationsdaten für Niederschlag 32

5.2.1 Thiessen-Polygone 32

5.2.2 Inverse-Distance-Weighting 34

5.2.2.1 Implementation im TRAIN-ZIN-Modell 36

5.3 Abflusskonzentration 37

5.3.1 Bisherige Situation 37

5.3.2 Weiterentwicklung des UH-Ansatzes 38

5.3.3 Übergang vom ZIN- zum Routing-Zeitschritt 43

5.4 Channel-Routing 43
5.4.1 Implementation des Muskingum-Cunge-Verfahrens 44
5.4.2 Längen der Gerinnesegmente 48
5.5 Warmstart 49
5.6 Wasserbilanz 49
5.7 Weitere programmtechnische Veränderungen 49
5.7.1 Einlesen von Grids 49
5.7.2 Ausgabe von Grids in Dateien 51
5.7.3 char-Arrays und strings 51
5.7.4 Unreferenzierter Programmcode 52
5.7.5 Lesen und Schreiben von Dateien 52
5.7.6 Schnittstelle zwischen Abflusskonzentration und Routing 54
5.7.7 Einlesen der Gerinneparameter 54
5.7.8 Datumsformat 54
5.7.9 Optionale Verwendung von TRAIN 54
5.8 Fazit 55

6 Das Dragonja-Gebiet 57
6.1 Lage und Klima 57
6.2 Geologie, Topografie und Böden 58

7 Eingangsdaten und Modellparameter 59
7.1 Vorhandene Daten und Datenaufbereitung 59
7.1.1 Pegel Dragonja 59
7.1.2 Niederschlagsdaten 60
7.1.3 Meteorologische Daten 62
7.1.4 Räumlich verteilte Daten 63
7.2 Parametrisierung 64

7.2.1 Einteilung der Teileinzugsgebiete 64
7.2.2 Boden- und Abflussbildungsparameter für das ZIN-Modell 65
7.2.3 Boden- und Landnutzungsparameter für das TRAIN-Modell 66
7.2.4 Gerinneparameter 67
7.3 Fazit 67

8 Ergebnisse der Modellierung 69
8.1 Der Nash-Sutcliffe-Index 69
8.2 Kalibrierungsphase 72
8.2.1 Verhalten des Routing 76
8.3 Validierungsphase 77
8.4 Warmlaufphase des Modells 77
8.4.1 Anpassung der Bodenfeuchte 77
8.4.2 Stabilisierung des Channel-Routing 79
8.5 Trockenwetterabfluss 79
8.5.1 Tagesgang des Abflusses 80
8.6 Fazit 82

9 Diskussion und Ausblick 85
9.1 Verarbeitung von Niederschlagsdaten 85
9.2 Abflusskonzentration 85
9.3 Routing 86
9.4 Abflussbildung 86
9.5 Mögliche Erweiterung des TRAIN-ZIN-Modells 87
9.6 Implementation des TRAIN-ZIN-Modells 88
9.6.1 Dokumentation 88
9.7 Weitere Modellierung im Dragonja-Gebiet 89

10 Literatur 91

10.1 verwendete Software: 94

Anhang A – Abbildungen 97

Anhang B – Tabellen 100

Anhang C – Quellcode 105

C.1 Klasse Controller 105

C.2 Klasse Grid 109

C.3 Klasse QConc 112

C.4 Klasse RainGauges 119

C.5 Klasse Routing 128

I Verzeichnis der Abbildungen

Abb. 4.1 Konzeption der Abflussbildung im ZIN-Modell 17

Abb. 4.2 Programmstruktur des TRAIN-ZIN-Modells 23

Abb. 5.1 Struktur der Controller-Klasse (Aufruf in umgekehrter Richtung der Rückgabe) 31

Abb. 5.2 Zwei Anordnungen mit gleichem IDW-Ergebnis für Punkt *P* (nach SHEPARD 1968) 35

Abb. 5.3 Übermäßige Gewichtung von Stations-Clustern mit dem IDW-Verfahren . . . 35

Abb. 5.4 Inverse-Distance-Grid für den 11.5.2002 (Tagessumme, Werte 0 bis 18 mm, Station Labor ohne Daten) 37

Abb. 5.5 Gamma-Verteilungen mit einer Konzentrationszeit von 30 bei verschiedenen Parameter-Paaren (α, β), Berechnung von β nach Gleichung 5.7 39

Abb. 5.6 Einfluss des Abfluss-Konzentrations-Parameters *a* 41

Abb. 5.7 Einfluss des Abfluss-Konzentrations-Parameters b 42

Abb. 5.8 Iterative Berechnung des Abflusses nach dem nicht-linearen Muskingum-Cunge-Verfahren 45

Abb. 5.11 Entscheidungsbaum für die Verwendung des TRAIN-Modells. (Bei positiver Entscheidung nach rechts, sonst nach links. *useTrain*: Verwendung von TRAIN ein/aus. *reTrain*: TRAIN jedenfalls neu durchlaufen. *file exists*: Datei mit Verdunstungswerten ist vorhanden. 0: Train wird nicht gestartet. 1: TRAIN wird gestartet) 55

Abb. 6.1 Lage des Dragonja-Gebiets (KEESTRA 2005) 57

Abb. 7.1 Verfügbarkeit von Daten (von unten: Niederschlag, weitere meteorologische Daten, Abfluss, alle Daten gleichzeitig) 59

Abb. 7.2 Lage der Niederschlagsstationen, der Pegel und der Meteorologischen Station (Hintergrund: Darstellung des DEM) 61

Abb. 7.3 Verfügbarkeit von Niederschlagsdaten der verschiedenen Stationen innerhalb der Simulationsperiode 62

Abb. 7.4 Verfügbarkeit meteorologischer Daten der Station Borst (von unten nach oben: Temperatur, relative Luftfeuchte, Windgeschwindigkeit, relative Sonnenscheindauer, alle Daten gleichzeitig) 63
Abb. 7.5 Einteilung der Teileinzugsgebiete und der Gerinnesegmente 64
Abb. 8.1 Modellergebnis für den Kalibrierungszeitraum Februar bis August 2002 70
Abb. 8.2 Modellergebnis im Validierungszeitraum. Farbige Graphen: mit verschiedenen Vorfeuchten modellierter Abfluss 71
Abb. 8.3 Modellergebnisse Januar bis März 2002 (Modellstartphase) 72
Abb. 8.4 Modellergebnisse April bis Mai 2002 73
Abb. 8.5 Modellergebnisse Mai bis Juni 2002 74
Abb. 8.6 Modellergebnisse Juni bis Juli 2002 75
Abb. 8.7 Modellergebnisse August 2002 76
Abb. 8.8 Verlauf der Angleichung drei verschiedener Modellläufe mit unterschiedlichen Vorfeuchten 78
Abb. 8.9 Modellergebnis bei Trockenwetter im Februar/März 2002 80
Abb. 8.10 Beziehung zwischen Strahlung, modellierter Verdunstung und Abfluss (März 2002) 81
Abb. 9.1 Einbindung eines zusätzlichen Speicherelementes in das ZIN-Modell 87
Abb. 10.1 trockenes Flussbett im Einzugsgebiet (Foto: Jens Lange) 97
Abb. 10.2 Brücke über die Dragonja nahe dem Pegel (Foto: Jens Lange) 97
Abb. 10.3 Flussbett im Dragonja-Eizugsgebiet (Foto: Jens Lange) 98
Abb. 10.4 Flyschwand im Dragonja-Einzugsgebiet (Foto: Jens Lange) 99

II Verzeichnis der Tabellen

Tabelle 3.1 Laufzeiten verschiedener Matrix-Implementationen 7

Tabelle 5.1 Beispiel für die Sortierung der Stationsindizes nach Entfernung zwischen Zelle und Station 33

Tabelle 6.1 Klimatische Bedingungen im Dragonja-Gebiet (nach MARDEŠIĆ ET AL., 1962) 58

Tabelle 7.1 Zuordnung der ursprünglichen Landnutzungsklassen zu TRAIN-Landnutzungsklassen 66

Tabelle 11.2 Bodenparameter im ZIN-Modell 100

Tabelle 11.3 Gerinnenetz-Datei (Teil 1/4, Schlüssel „RiverNet” in der Steuerungsdatei) . 101

Tabelle 11.4 Gerinnenetz-Datei (Teil 2/4) 102

Tabelle 11.5 Gerinnenetz-Datei (Teil 3/4) 103

Tabelle 11.6 Gerinnenetz-Datei (Teil 4/4) 104

Zusammenfassung

Ziel dieser Arbeit war es, das Niederschlags-Abfluss-Modell ZIN mit dem daran gekoppelten Verdunstungsmodell TRAIN zu überarbeiten und zu erweitern und durch eine Anwendung im Dragonja-Einzugsgebiet in Süd-West-Slowenien zu testen.

Zur Verbesserung der Bedienbarkeit des Modells wurde eine Benutzerschnittstelle implementiert, die auf in einer Textdatei gespeicherten Schlüssel-Werte-Paaren beruht. Die Schnittstelle ermöglicht nun auch Benutzern ohne Kenntnisse der Programmiersprache C++ eine Verwendung des Modells. Das Aufstellen und Verändern eines Modell-Setups wird vereinfacht und beschleunigt, während mehr Sicherheit bezüglich der Konsistenz eines Modell-Setups gewährleistet wird.

Zur Verarbeitung von Niederschlags-Stationsdaten wurden das Thiessen-Polygon- und das Inverse-Distance-Weighting-Verfahren (IDW) implementiert. Für das IDW-Verfahren ist eine Höhenkorrektur der Niederschläge über die Angabe eines relativen Gradienten möglich. Die Zahl der maximal zu verwendenden Stationen lässt sich vom Benutzer festlegen.

Im TRAIN-ZIN-Modell erfolgt die Abflusskonzentration nach dem Unit-Hydrograph-Ansatz (UH). Möglich ist die Verwendung einer auf Messungen basierenden Konzentrationskurve für alle Teileinzugsgebiete, alternativ dazu kann ein synthetischer UH verwendet werden. Für den synthetischen UH wurde die Verwendung einer EV-I-Verteilung implementiert, die einen der Konzentrationszeit entsprechenden Parameter enthält und einen Skalierungsfaktor zur Kontrolle der Breite der Verteilung. In der Modellierung zeigte sich, dass die beiden Parameter eine gute Möglichkeit zur Feinabstimmung des zeitlichen Eintreffens einer Hochwasserwelle am Gebietsauslass sowie der Auflösung zwischen kurz aufeinanderfolgenden Niederschlagsinputs bieten. Die zusätzlich eingeführte automatische Anpassung der Parameter an unterschiedliche Teileinzugsgebiete anhand von mittlerer Hangneigung und Einzugsgebietsgröße hatte keinen erkennbaren Einfluss auf die Qualität der Modellergebnisse.

Das Channel-Routing im ZIN-Modell verwendet das nicht-lineare Muskingum-Cunge-Verfahren (MC). Das Abbruchkriterium der MC-Iteration wurde so umformuliert, dass es unabhängig von einer absoluten Abflussmenge ist und so für alle Segmente und Abfluss-Zustände die Verwendung des nicht-linearen Verfahrens garantiert. Im Zuge dessen wurden auch die lateralen Zuflüsse sowie Zuflüsse aus anderen Gerinnesegmenten neu angeordnet. Die Modellergebnisse zeigten, dass einige der vorher aufgetretenen Probleme bezüglich der Stabili-

tät beseitigt werden konnten. Bei sehr steil ansteigenden Abflüssen zeigt sich ein unrealistisches, verfahrensbedingtes vorheriges Abfallen der Werte.

Die weitere Bearbeitung des Modells umfasste unter anderem die Einführung einer Warmstart-Option, bei der für die Bodenfeuchte entweder ein globaler Wert oder flächenverteilte Werte über ein Grid verwendet werden können. Die Ausgabe der Wasserbilanz wurde vervollständigt, sodass Aussagen über einzelne Glieder des Wasserhaushalts getroffen und Modellfehler leichter erkannt werden können.

Zahlreiche weitere programmtechnische Änderungen dienten vorwiegend der Verbesserung des Bedien-Komforts, der Sicherheit im Bezug auf das Vermeiden oder Erkennen von internen Fehlern wie unzulässigen Speicherzugriffen oder unzulässigen Eingabedaten, der Straffung des Quellcodes und der Verkürzung der Laufzeit.

Das Dragonja-Gebiet liegt nach der Einteilung nach THORNTHWAITE (1948) im Bereich der humiden Klimate und unterscheidet sich damit stark vom ursprünglichen Anwendungsbereich des ZIN-Modells in ariden Gebieten. Die Modelleffizienz für den Kalibrierungszeitraum von Februar bis August 2002 ergab eine Modell-Effizienz nach NASH & SUTCLIFFE (1970) von -0,44 und war damit nicht zufriedenstellend. Dass die Modelleffizienz in der Validierungsphase mit 0,63 deutlich besser war, hat durch die Kürze dieser Phase von lediglich drei Wochen eine geringe Aussagekraft und mag auch daran liegen, dass hier über den vorhergehenden Basisabfluss angepasste Vorfeuchte verwendet wurde.

Die im Modell dargestellten Abflussbildungsprozesse von Infiltrations- und Sättigungsüberschuss scheinen nicht auszureichen, um die im Dragonja-Gebiet real auftretenden Abflussbildungsprozesse wiederzugeben. Möglicherweise könnte hier bereits die Einführung einer Vorfeuchteabhängigkeit der Infiltrationsrate eine Verbesserung bringen. Zur Verbesserung der Modellierung des Basisabflusses scheint die Einführung eines zusätzlichen tiefen Speicherelements sinnvoll.

Stichworte: Dragonja, TRAIN, ZIN, Niederschlags-Abfluss-Modell, Inverse-Distance-Weighting, Muskingum-Cunge

Summary

The objective of this study was to revise the rainfall-runoff-Model ZIN with the coupled evaporation-model TRAIN, to extend its functionality and to test it in an application to the Dragonja study area in south-west Slovenia.

To improve the usability of the model, a user-interface was implemented. It is based on key-value pairs that the user stores in a text file. The interface enables users without any knowledge of the programming language C++ to use the model. Also creating and changing model setups is facilitated and becomes less time consuming while it is easier to maintain consistency of a model setup.

To permit usage of rain gauges as a source of precipitation-input the Thiessen-polygon-method, also known as Voronoi- or Dirichlet tessellation, and the Inverse-Distance-Weighting-method (IDW) were implemented. The IDW features an optional altitude-correction of precipitation based on a relative gradient. The number of stations to be used can be limited to save computational time.

Runoff concentration in the model is being represented by the Unit-Hydrograph (UH) approach. The application of a unique curve based on field measurements to all sub-catchments is possible. Alternatively a synthetic UH can be used. For this the usage of the EV-I-distribution was implemented. The EV-I has one parameter corresponding to concentration time and one controlling temporal extension. During modelling both parameters proved to be a convenient tool for fine tuning of the arrival time of a flood at the gauge and the separation of precipitation events of short succession. Additionally, an automated adaptation of the parameters for individual sub-catchments based on average hillslope and catchment size was incorporated, but did not seem to have any positive effect on modelling results.

Channel routing in the ZIN-Model is based on the non-linear Muskingum-Cunge (MC) scheme. The stop criterion of the MC iteration has been changed to be independent from an absolute discharge value. By using a relative formulation instead, usage of the non-linear scheme is ensured throughout different discharge conditions in all channel segments. In the course of this the alignment of lateral- and tributary inflow has been rearranged as well. Model results showed that some of the stability problems arising in earlier applications could be eliminated. With some steeply rising hydrographs the model still shows a sharp drop at the beginning which is thought to be inherent to the MC-scheme.

Further developments of the model included a hotstart option using either a global value for initial moisture or a grid with a value for every grid cell. Output of the water balance was refined by including missing terms. Thus users can check the included terms of the water balance and an additional plausibility check of the model is possible.

A number of further technical changes was made to improve usability, safety concerning error recognition and -handling as well as illegal memory accesses and illegal input data. The source code was condensed and run time reduced.

According to the Thornthwaite-classification (THORNTHWAITE 1948) the Dragonja-catchment belongs to the humid zone. This makes it very different from the arid zones the ZIN-model was first developed for. Model efficiency calculated by the Nash-Sutcliffe-index (NASH & SUTCLIFFE 1970) for the calibration period February 2002 to August 2002 yielded an unsatisfactory value of -0.44. Model efficiency for the validation period was 0.63. Due to the short duration of validation of only three weeks this is of limited significance. Also the adaptation of initial moisture using initial baseflow-values may be a reason for this good result.

It seems that the model´s runoff generation processes of infiltration excess and saturation excess overland flow can not sufficiently represent natural runoff generation processes in the Dragonja catchment. A further step to enhance this process representation could be to introduce a moisture dependency of the infiltration rate. To improve baseflow representation the inclusion of an additional deep storage into the model should be considered.

keywords: Dragonja, TRAIN, ZIN, rainfall-runoff model, Inverse Distance Weighting, Muskingum-Cunge

1 Einleitung

Das ZIN-Modell ist ein von LANGE (1999) als unkalibriert konzipiertes Niederschlags-Abfluss-Modell für Einzelereignisse in großen ariden Gebieten. Seitdem folgten Anwendungen im semi-ariden, mediterranen und sogar im humiden Raum am Kaiserstuhl bei Freiburg. Dabei wurde das Modell ständig weiter bearbeitet, mit neuen Funktionen und besserer Repräsentation der ablaufenden Prozesse ausgestattet und an die jeweiligen Modellierungserfordernisse angepasst. Die jüngere Entwicklung umfasst die Berechnung der Sickerung aus einer Bodenzelle nach Van Genuchten, die bei Bedarf als Basisabfluss oder Interflow interpretiert werden kann sowie die Kopplung mit dem Verdunstungsmodell TRAIN. Dies sind wichtige Schritte zur Anwendung des Modells in feuchteren Klimaten sowie zur Anwendung über die Ereignis-Zeitskala hinaus.

Das Dragonja-Gebiet in Slowenien ist den humiden Klimaten zuzuordnen. Mit bis zu 1200 mm jährlichen Niederschlags in den Hochlagen unterscheidet es sich wesentlich von den Gebieten, für die das ZIN-Modell ursprünglich konzipiert wurde.

2 Zielsetzung und Vorgehensweise

Ziel dieser Arbeit war zunächst die Weiterentwicklung des TRAIN-ZIN-Modells. Dazu zählt

1. die Erweiterung der Fähigkeiten des Modells, natürliche Abläufe darzustellen

Dazu gehört

   a. die Erweiterung um die Möglichkeit, Niederschlagsdaten in Form von Stationsdaten zur Verfügung zu stellen, die innerhalb des Modells verarbeitet werden,
   b. die Weiterentwicklung des synthetischen Unit-Hydrograph-Ansatzes für den Abfluss aus den Teileinzugsgebieten und
   c. die Überarbeitung der Implementation des Channel-Routing.

2. die Verbesserung der Interaktion des Anwenders mit dem Modell und ein verbesserter Umgang mit Ein- und Ausgabedaten
3. eine Überarbeitung des Quellcodes zur Verbesserung der Programm-Sicherheit bzgl. unerwarteter Eingaben oder Ereignisse, der Geschwindigkeit, des Speicherbedarfs sowie der korrekten Handhabung der C++-Sprachelemente

Die Entwicklung des Modells ZIN begann mit einzelnen Modulen für die verschiedenen in die Modellierung einbezogenen Prozesse. Nach und nach wurden die Komponenten ausgebaut und schließlich zu einem integrierten Modell zusammengefügt. Mit steigendem Umfang des Modellcodes wurde die Handhabung zunehmend umständlicher, da alle Informationen über das Modell-Setup direkt (und häufig mehrfach) im Quellcode eingegeben wurden. Ein weiteres Ziel dieser Arbeit war daher die Erstellung eines Benutzer-Interface, sodass Quellcode und Modell-Setup-Informationen getrennt werden können und für die Anwendung des Modells keine Änderungen am Quellcode nötig sind.

Schließlich sollte das Modell in einer Anwendung im Dragonja-Gebiet in Slowenien getestet werden.

2.1 Aufbau der Arbeit

Zunächst werden einige Besonderheiten der Programmiersprache C++ und der Entwicklungsumgebung *Microsoft Visual Studio* beschrieben, die bei der bisherigen Entwicklung des Programms von Bedeutung waren. Es folgt eine Beschreibung der Modelle TRAIN und ZIN sowie

ihrer Komponenten. Im nächsten Kapitel werden die am Modell vorgenommenen Änderungen beschrieben.

Die Anwendung des Modells im Dragonja-Gebiet ist gegliedert in die Beschreibung des Gebietes, eine Beschreibung der verwendeten Input-Daten und Parameter sowie die Ergebnisse der Modellierung.

Abschließend wird ein Ausblick auf Möglichkeiten und Bedarf bei der weiteren Entwicklung des TRAIN-ZIN Modells sowie weitere Modellierung des Dragonja-Gebietes gegeben.

Im Anhang finden sich neben Abbildungen und Tabellen Auszüge aus dem im Zuge dieser Arbeit verfassten Quellcode. Die neu erstellten Klassen sind vollständig wiedergegeben, während von den anderen Klassen nur die Teile einbezogen wurden, die zumindest stark überarbeitet wurden. Einige Teile des bearbeiteten Quellcodes wurden wegen ihrer Trivialität nicht aufgenommen. Eine vollständige Version des TRAIN-ZIN-Modells in der zum Abschluss dieser Arbeit aktuellen Version findet sich in der zu dieser Arbeit gehörenden Datendokumentation.

Zur Kennzeichnung von C++-Sprachelementen und Variablennamen wird Text in der Schriftart `Courier New` gesetzt, um ihn vom übrigen Fließtext abzugrenzen. Längere Abschnitte von Quellcode enthalten außerdem eine farblich unterschiedliche Kennzeichnung nach ihrer Funktion. Schlüsselwörter sind blau dargestellt (`while`), Operatoren rot (`==`), Kommentare grün (`// explain...`) und strings türkis (`"error!"`). Variablen-, Methoden- und Klassennamen bleiben schwarz.

3 Werkzeuge der Programmierung

3.1 Die Programmiersprache C++

Die Programmiersprache C++ wurde in den Jahren 1980-1983 von Bjarne Stroustrup entwickelt. Sie ist eine Weiterentwicklung von C, das vollständig darin enthalten ist (STROUSTRUP 1997). Ihr wichtigstes Unterscheidungsmerkmal gegenüber C ist die Unterstützung von objektorientierter Programmierung. In diesem Abschnitt wird nur auf einige Aspekte der Programmiersprache eingegangen, die für die bisherige Entwicklung der TRAIN-ZIN-Software von besonderer Bedeutung waren.

3.1.1 C-Arrays und C++ Container-Klassen

Um Felder von Daten gleichen Typs anzulegen, gibt es in C Arrays – in C++ stehen dafür sogenannte Container-Klassen zur Verfügung. Diese können Felder von beliebigen, auch selbst definierten Typen enthalten, bieten zusätzliche Funktionen und eine größere Sicherheit.

Bei Element-Zugriffen auf C-Arrays findet keinerlei Überprüfung statt, ob der verwendete Index im zulässigen Bereich liegt. Dies gilt sowohl für Lese- wie auch für Schreibzugriffe – die Verantwortung für die richtige Handhabung liegt vollständig beim Programmierer. Erfolgt ein Zugriff außerhalb des zulässigen Bereiches, wird auf Speicherbereiche zugegriffen, die nicht für diese Verwendung vorgesehen sind und das Ergebnis ist nicht vorhersehbar.

Bei Lesezugriffen kann dies zu einem unmittelbaren Abbruch des Programms führen, beispielsweise wenn in dem zufällig gelesenen Speicherbereich eine 0 steht und versucht wird, durch diese zu teilen. Der zufällig gelesene Speicherbereich kann allerdings auch einen Wert enthalten, der weiterverarbeitet werden kann. In diesem Fall ist den Ergebnissen des Programms zu misstrauen.

Bei Schreibzugriffen können die Folgen ebenso fatal sein. Im besten Fall wird der beschriebene Speicherbereich nicht anderweitig verwendet und die geschriebene Information richtet keinen Schaden an. Unter Umständen ist sie später sogar wieder verwendbar, wenn der Speicher nicht zwischenzeitlich anderweitig verwendet wird, allerdings bleibt dies dem Zufall überlassen. Der Versuch, auf geschützte Speicherbereiche anderer Programme zuzugreifen, kann zu einer `Exception` (Ausnahme) führen, in diesem Fall ist der Fehler relativ leicht zu lokalisieren. Anders verhält es sich, wenn erfolgreich auf den Speicher anderer Elemente desselben Programms oder auf Speicher anderer Programme geschrieben wird. Die Folgen können von

unerklärlichem Verhalten irgendeines Programms bis hin zum Absturz des gesamten Systems reichen.

Die Verwendung der „unsicheren" C-Typen lässt sich in C++ in aller Regel vermeiden. So existiert in C++ als Ersatz für die Verwendung von `char`-Arrays als Repräsentation von Strings die `string`-Klasse. Vorteile sind unter anderem eine intuitive Verwendung der Operatoren `+`, `=` und `==`, die veränderbare Länge sowie die Vermeidung eines unbemerkten Zugriffs auf einen Bereich außerhalb des Strings, was durch die Member-Funktionen `size()` und `length()` erleichtert wird, über die jederzeit Informationen über die aktuelle Länge des `strings` abrufbar sind.

Ähnlich ist es bei den Zahlen-Arrays, die sich häufig durch Objekte der Klasse `vector` ersetzen lassen. (Für manche Anwendungen sind die verschiedenen Listen-, Schlangen- oder Stapel-Klassen besser geeignet.) Vektoren sind ebenfalls dynamische Objekte, die, anders als Arrays, die Information über ihre Größe selbst enthalten und außerdem zahlreiche hilfreiche Member-Funktionen zur Verfügung stellen, die für Arrays in der Regel selbst implementiert (und getestet) werden müssen.

3.1.2 Laufzeit-Tests

Ein Nachteil von Vektoren ist die langsamere Zugriffsgeschwindigkeit im Vergleich zu C-Arrays. Da der Zugriff auf eine große Menge von in Grids organsisierten Daten die Laufzeit des TRAIN-ZIN Modells wesentlich mitbestimmt, wurden Laufzeit-Messungen für die folgenden verschiedenen Implementationen zweidimensionaler Matrizen durchgeführt:

1. zweidimensionaler Vektor (Typ `vector<vector<int>>`)
2. eindimensionaler Vektor (Typ `vector<int>`), bei dem die Position *(x, y)* an der Stelle *(y * xSize + x)* liegt mit *xSize* = Größe der Matrix in x-Richtung
3. zweidimensionales C-Array (Typ `int**`)
4. eindimensionales C-Array (Typ `int*`) analog zum eindimensionalen Vektor, dies ist die in TRAIN-ZIN bisher verwendete Implementation für Grids mit Gebietsdaten

Eine Messung der Laufzeit für 200 Durchläufe mit je einem Schreib- und einem Lese-Zugriff auf jedes Element einer 600 x 1200-Matrix (ca. $2{,}9 \cdot 10^8$ Operationen) auf einem 800-Mhz-Prozessor ergab für zwei unterschiedliche Konfigurationen die in Tabelle 3.1 dargestellten Ergebnisse:

Tabelle 3.1 Laufzeiten verschiedener Matrix-Implementationen

Typ	release-Modus, Laufzeit in s	debug-Modus, Laufzeit in s
2D-Vektor	5,38	342
1D-Vektor	3,21	165
2D-Array	3,77	7,79
1D-Array	1,47	3,98

Während die absoluten Werte eines solchen Tests nicht sehr aussagekräftig sind, da sie sowohl von der verwendeten Hardware als auch von den konkreten Beispieldaten abhängen können (FELLEISEN ET AL. 2001), lassen sich aus den Verhältnissen zueinander Schlüsse ziehen. Im debug-Modus benötigt die Vektor-Implementation im Vergleich zur entsprechenden Array-Implementation jeweils etwa das 40-fache an Zeit. Hier spielt sicher die Überprüfung der Gültigkeit von Vektoren-Indizes im debug-Modus eine Rolle, die bei Arrays nicht stattfindet. Im release-Modus schrumpft dieser Unterschied etwa auf den Faktor 1,4 (2D) bzw. 2,2 (1D). Eindimensionale C-Arrays stellen damit die schnellste der getesteten Implementationen einer 2D-Matrix dar.

Die Laufzeiten des debug-Modus liegen für die Arrays zwei- bis dreimal so hoch wie im release-Modus, bei den Vektoren liegt der Unterschied etwa beim Faktor 50-60. Um die Größenordnung der Bedeutung dieser Unterschiede auf das TRAIN-ZIN-Modell abzuschätzen, wurde ein Modelllauf über einen Tag ohne das TRAIN-Modell durchgeführt. Dabei ergab sich eine Laufzeit von 4:12 Minuten im release-Modus und eine Laufzeit von 14:00 Minuten im debug-Modus auf einem 3,2 Ghz Pentium-D Prozessor, was einem Verhältnis von 10:3 entspricht.

3.1.3 Memory-Leaks und Destruktoren

Anders als viele andere Programmiersprachen besitzt C++ kein umfassendes System der „garbage collection", also zur „Entsorgung" nicht mehr benötigter Objekte. Objekte, die in der Form

```
myClass object = myClass(param);
```

erzeugt werden, werden vom System verwaltet. Es ist aber auch möglich, Objekte unter Verwendung des Schlüsselwortes `new` zu erzeugen:

```
myClass object = new myClass(param);
```

Bei Arrays, deren benötigte Größe zur Kompilierungszeit nicht bekannt ist, sondern erst zur Laufzeit festgelegt werden soll, behilft man sich durch eine Liste von Zeigern auf den gewünschten Typen, für die ebenfalls Speicher mit dem Schlüsselwort `new` angefordert wird:

```
double* myArray = new double[size];
```

In diesen Fällen muss der Programmierer die Verwaltung des Objektes bzw. Arrays selbst übernehmen und auch für die ordnungsgemäße Freigabe verwendeter Ressourcen sorgen. Dazu gehört das explizite Freigeben von Speicher, wenn er nicht mehr benötigt wird. Wird dies versäumt, entstehen sogenannte Memory-Leaks: Speicher, der noch reserviert ist, also nicht neu vergeben werden kann, auf den aber nicht mehr zugegriffen werden kann.

Das Freigeben von Speicher geschieht durch den Aufruf des `delete`-Operators (bzw. `delete[]` für Arrays) am Ende des Gültigkeitsbereichs sowie für Objekte zusätzlich durch das Bereitstellen des richtigen Destruktors. Der Destruktor wird durch den `delete`-Operator implizit aufgerufen, ein expliziter Aufruf etwa der Form `~myClass()` sollte nicht erfolgen. Der Destruktor einer Klasse enthält wiederum den Aufruf der für die vorhandenen Instanzvariablen nötigen `delete`- oder `delete[]`-Operatoren.

Im folgenden Beispiel wird Speicher innerhalb einer Methode reserviert, aber nicht wieder freigegeben. Außerdem wird ein lokaler *Zeiger* auf das angelegte Array zur aufrufenden Stelle zurückgegeben. Dies wirft zusätzlich das Problem auf, dass über einen Zeiger weiterhin auf die Variable `lokaleVar` zugegriffen wird, obwohl deren Gültigkeitsbereich (nämlich „`methode`") verlassen wurde:

```
//...
int* eineVar = methode();
// weitere Operationen...

int* methode(){
int* lokaleVar = new int[someSize];
    // weitere Operationen
```

```
    return lokaleVar;
}
```

An dieser Stelle ist es nicht möglich, statt des Zeigers auf die lokale Variable eine Kopie des gesamten Arrays zurückzugeben, außerdem hat die Übergabe von Zeigern oder Referenzen bei großen Datenmengen den Vorteil einer deutlich höheren Geschwindigkeit. Die Speicherverwaltung muss dann aber auf der aufrufenden Seite erfolgen, z. B. in der Form

```
//...
    int* eineVar = new int[someSize];
    methode(eineVar);
    // weitere Operationen...
    delete [] eineVar;
    eineVar = 0;
}

void methode(int* varZeiger){
    // Operationen auf varZeiger
}
```

Hier wird von der aufrufenden Seite ein Zeiger übergeben. Alle Operationen, die in `methode` auf dem Zeiger `varZeiger` ausgeführt werden, betreffen also tatsächlich auch die Variable `eineVar`. Dieses Vorgehen ist z. B. analog den C-Funktionen zur Behandlung von C-Strings, die ja `char`-Arrays sind, wie z. B. `strcpy(char*, char*)`, bei denen der zu verändernde Wert nicht von der Funktion zurückgegeben, sondern als Parameter übergeben wird.

3.1.4 Mehrfach-Verwendung von Code

Einer der großen Vorteile von Computern ist, dass man Sachen, die man einmal in eine dem Rechner verständliche Form gebracht hat, nie wieder selbst ausführen muss (HELMKE ET AL. 2007). Dies bedeutet auch, dass es in der Regel nicht nötig ist, den gleichen Algorithmus (oder gleiche Teile eines Algorithmus) mehrfach zu implementieren. Moderne Programmiersprachen arbeiten daher mit Prozeduren, Funktionen oder Methoden.

Gleichen oder nahezu gleichen Quellcode an verschiedenen Stellen eines Programms zu verwenden, führt zu einer schlechteren Übersichtlichkeit sowie besonders bei umfangreichem Code zu stark erschwerter Wartung, da alle Änderungen oder Fehlerkorrekturen an mehreren Stellen durchgeführt werden müssen. In der Regel ist es möglich, den Quellcode so zu strukturieren, dass längere Code-Teile nur einmal verwendet werden. Dazu können z. B. Teile, die an mehreren Stellen gebraucht werden, als eigene Methode „ausgelagert" werden. Eine Methode, die sich durch zusätzliche Funktionalität von einer anderen unterscheidet, sollte nur diese zusätzliche Funktionalität enthalten und für die Lösung des gleichen Anteils auf die einfachere Methode zurückgreifen.

3.2 Die Entwicklungsumgebung Microsoft Visual Studio

Microsoft Visual Studio (VS) ist eine integrierte Entwicklungsumgebung für die Programmiersprachen Visual Basic, C# und C++. Unter anderem sind außerdem die Bibliotheken C++/CLI enthalten, die die komfortable Erstellung grafischer Windows-Programme erlauben. Da sich die Syntax dafür allerdings grundlegend von der C++-Syntax unterscheidet, wurde diese nicht verwendet. (Siehe auch Abschnitt 5.1.1)

VS umfasst unter anderem einen Texteditor mit Syntaxhighlighting, einen Compiler, der den Quelltext in Maschinensprache übersetzt und einen Debugger, mit dessen Hilfe der Programmablauf quelltextbezogen verfolgt werden kann, um Fehler aufzuspüren.

3.2.1 Solutions und Projekte

Der Programmcode wird von Visual Studio in einfachen Textdateien abgespeichert. Für die C++-Dateien haben diese die Dateinamen-Erweiterung .cpp für die Dateien mit dem eigentlichen Code und .h für die header-Dateien. Dabei enthält jedes Paar von .cpp/.h Dateien den Code für eine Klasse. Die zusammengehörigen Klassen können als Projekt gegliedert werden, weiterhin können mehrere Projekte in einer sogenannten „Solution" zusammengefasst werden. Das TRAIN-ZIN - Modell besteht aus einer Solution mit dem C++-Projekt, das den ZIN-Teil sowie das Interface zum TRAIN-Modell enthält und dem Fortran-Projekt, das den größten Teil des TRAIN-Modells beinhaltet.

Für die Solution sowie für jedes Projekt lassen sich verschiedene Eigenschaften festlegen. Um das Modell betreiben zu können, muss z. B. bei den ZIN-Eigenschaften unter „Debuggen"

im Eintrag „Befehlsargumente" der Pfad zur Steuerungsdatei (.ctr) eingetragen werden, der dann beim Start des Programms als Argument-String übergeben wird.

3.2.2 Debugging

Zur Verfolgung des Ablaufs eines Programms sowie insbesondere zur Fehlersuche steht in Visual Studio ein Debugger zur Verfügung. Damit lassen sich die Werte von Variablen während der Laufzeit kontrollieren und der Programmfluss an jeder Stelle vorübergehend unterbrechen. Da eine Überwachung für die C++-Container-Klassen `vector` und `string` erst seit der der verwendeten Programmversion 2003 nachfolgenden Version unterstützt wird, wurde zusätzlich das add-in „VSE Debug" verwendet.

3.3 Zusammenarbeit mehrerer Entwickler

Jede Anwendung des Modells ging bisher auch mit einer mehr oder weniger starken Modifizierung des Modells einher. Davon hat das Modell einerseits profitiert, da es immer wieder in die Lage versetzt wurde, neue Gegebenheiten des zu modellierenden Gebietes zu berücksichtigen. Andererseits liegt hier ebenso die Gefahr, dass Teile des Modells wieder verloren gehen, wenn sie von einem Anwender nicht gebraucht und daher deaktiviert werden. So beschreibt THORMÄHLEN (2003) die Verwendung von variablen Infiltrationsraten für die Abflussbildung. In der Version von 2007 ist diese Möglichkeit nicht mehr vorhanden. Arbeiten mehrere Personen gleichzeitig an der Software, entstehen außerdem verschiedene Versionen des Modells. Neben dem Erkennen der Unterschiede zwischen zwei Versionen einer Quellcode-Datei ist es entscheidend, ob ein Unterschied durch das Löschen in der einen oder dem Hinzufügen in der anderen Version zustande kommt. Dadurch wird das Zusammenführen verschiedener Versionen erheblich erschwert.

Außerdem ist es wünschenswert, Zwischenstände der Entwicklung zu speichern. Diese dienen einerseits als Sicherung gegen Datenverlust, andererseits kann es nötig sein, auf ältere Versionen zurückzugreifen, um Vergleiche anzustellen oder für den Fall, dass sich eine spätere Bearbeitung als nicht sinnvoll herausstellt und rückgängig gemacht werden soll.

Schon in der Anfangsphase dieser Arbeit wurde deutlich, dass ein manueller Abgleich dieser unterschiedlichen Versionen, selbst wenn er häufig durchgeführt wird, nicht praktikabel ist. Der Zeitaufwand zum Versionsabgleich war annähernd so groß, wie der Aufwand zur Programmie-

rung selbst. Auch die Speicherung von Zwischenständen der Entwicklung ist mit einigem Aufwand verbunden.

Da dieses Problem überall auftritt, wo in Gruppen an Software oder anderen Projekten gearbeitet wird (HELMKE ET AL. 2007), gibt es eine Vielzahl von Programmen, die diesen als „Versionierung" bezeichneten Vorgang weitgehend automatisieren. Auf den Rechnern der an der TRAIN-ZIN-Entwicklung beteiligten Personen wurde dafür die weitverbreitete Open-Source-Software „Subversion" (SVN) installiert zusammen mit dem Client „Tortoise", der eine einfache grafische Benutzung aus dem Windows-Explorer heraus ermöglicht.

Tortoise-SVN arbeitet nicht nach dem strikten *lock-modify-write*-Verfahren, bei dem eine Datei zum bearbeiten für die anderen Teilnehmer gesperrt wird, sondern nach dem *copy-modify-merge*-Ansatz (HELMKE ET AL. 2007). Dabei existiert ein für alle Beteiligten zugängliches Verzeichnis, das als „Repository" bezeichnet wird. Dort sind eine Anfangsversion und alle bisherigen Veränderungen gespeichert, was auch ermöglicht, auf jeden bisherigen Entwicklungsstand zurückzugreifen. Jeder Entwickler arbeitet auf seinem Rechner mit einer Arbeitskopie. Von Zeit zu Zeit sendet er die an dieser Arbeitskopie vorgenommenen Änderungen an das Repository. Andere Entwickler können diese Änderungen abrufen und in ihre Arbeitskopie integrieren. Wurde die gleiche Datei von verschiedenen Personen bearbeitet, werden in den meisten Fällen die Änderungen beider Personen automatisch integriert. Ist dies nicht möglich, entsteht ein Bearbeitungskonflikt, der manuell geklärt werden muss. Zum manuellen Vergleich wurde die freie Software WinMerge verwendet, die sich in Tortoise-SVN integrieren lässt und Syntaxhighlighting für C++ bietet.

3.4 Fazit

C++ ist eine mächtige Programmiersprache, die dem Programmierer viele Freiheiten lässt und eine systemnahe Programmierung unterstützt. Mehr als bei vielen anderen Hochsprachen bringt diese Freiheit aber auch die Gefahr mit sich, fehlerhaften Code zu produzieren. Einige der häufigsten Fehlerquellen wurden in diesem Abschnitt beschrieben. Außerdem wurde in einem exemplarischen Vergleich mehrerer Matrix-Implementationen dargestellt, welchen Einfluss unterschiedliche Implementation gleicher Funktionalität auf die Geschwindigkeit haben kann und dass die besonders geschwindigkeitskritische Matrix-Repräsentation im ZIN-Modell die schnellste unter den getesteten Varianten ist. Gleichzeitig konnte die Bedeutung der in VS verfügbaren unterschiedlichen Kompilationsmodi „debug" und „release" auf die praktische

Verwendbarkeit eines rechenintensiven Programms wie dem TRAIN-ZIN-Modell gezeigt werden.

4 Beschreibung des TRAIN-ZIN-Modells

4.1 Das ZIN-Modell

Das Modell ZIN ist ein ursprünglich für die ereignisbezogene Anwendung in großen, ariden Gebieten erstelltes Niederschlags-Abfluss-Modell, das von LANGE 1999 vorgestellt wurde. Da in ariden Gebieten die Datenlage der gemessenen Abflüsse häufig schlecht ist (LANGE 1999), wurde das Modell als unkalibriert konzipiert. Eine umfassende Untersuchung des Gebietes soll dabei die Kalibrierung ersetzen.

LANGE (2001) nennt drei Voraussetzungen für die Verwendung nicht-kalibrierter Modelle für Flash-Floods in der ariden Zone:

1. unmittelbare Reaktion des Gebietes auf einen Niederschlagsinput mit zu vernachlässigenden Vorereignis-Komponenten
2. Dominanz von Oberflächenprozessen, durch Felduntersuchungen belegt und quantifiziert
3. Verfügbarkeit von zeitlich und räumlich hoch aufgelösten Niederschlagsdaten

Seit der Erstellung des Modells wurde sein Anwendungsbereich im Zuge von Modellanwendungen erweitert. SCHÜTZ (2006) führte für die Anwendung in einem mediterranen Einzugsgebiet den Prozess des Sättigungsflächenabflusses ein, GUNKEL (2007) band das Verdunstungsmodell TRAIN ein, GAßMANN (2007) entwickelte ein einfaches Basisabfluss-Modul, das in der vorliegenden Arbeit verwendet wurde, um die im humiden Dragonja-Gebiet zu erwartenden Interflow-Prozesse abzubilden.

Weiterentwicklungen erfolgten unter anderem auch durch WAGNER (2002), THORMÄHLEN (2003), LEISTERT (2005) und FISCHER (2007).

4.1.1 Niederschlag

In den bisherigen Versionen arbeitete das Modell entweder mit außerhalb des Modells erzeugten Niederschlagsgrids (z. B. aus Niederschlags-Radar) oder räumlich homogenem Niederschlag. Die Verarbeitung von Niederschlagsstationen wurde im Zuge dieser Arbeit hinzugefügt und wird ausführlich in Abschnitt 5.2 beschrieben.

4.1.2 Verdunstung

Tageswerte der Verdunstung werden räumlich verteilt vom TRAIN-Modell berechnet (siehe Abschnitt 4.2). Das TRAIN-Modell erhält dafür als Input neben meteorologischen-, Boden- und Landnutzungsparametern den vom ZIN-Modell berechneten Bodenwassergehalt als prozentuale Füllung des Bereichs zwischen permanentem Welkepunkt und der Feldkapazität:

$$Bodif = \frac{\theta_{act} - \theta_{PWP}}{\theta_{FK} - \theta_{PWP}} \qquad (4.1)$$

Bodif: von TRAIN verwendeter Bodenwassergehalt

θ_{act}, θ_{PWP}, θ_{FK}*: Bodenwassergehalt aktuell, beim PWP und bei FK*

Die Tageswerte werden unter Verwendung der stündlichen Strahlungswerte und des Niederschlags auf die einzelnen ZIN-Zeitschritte verteilt. Aus der Strahlung werden Gewichte für die Stunden des Tages berechnet, die Nachtstunden ($G < 5$ W/m²) sowie Stunden, in denen Niederschlag fällt, werden dabei explizit ausgeschlossen. Innerhalb der verbliebenen Stunden wird die Verdunstung gleichmäßig auf die Zeitschritte verteilt.

4.1.3 Abflussbildung

Die Abflussbildung im ZIN-Modell stellt die in ariden Gebieten dominanten Prozesse der Abflussbildung, Oberflächenabfluss durch Infiltrationsüberschuss sowie durch Sättigungsüberschuss dar. Außerdem besteht die Möglichkeit, die Sickerung aus den Bodenzellen dem Abfluss hinzuzufügen. Die zum Basisabfluss beitragende Fläche kann durch ein Grid festgelegt werden.

Das Konzept der Abflussbildung ist in Abbildung 4.1 dargestellt. In einem Zeitschritt gefallener Niederschlag wird zunächst dem Anfangsverlustspeicher hinzugefügt. Dieser kann als eine Kombination von Interzeptions- und Muldenspeicher interpretiert werden. Verdunstung findet zunächst aus diesem Anfangsverlustspeicher statt, soweit dessen Füllung das zulässt. Weitere Evapotranspiration findet direkt aus dem Bodenspeicher statt.

Ist die Bilanz aus Niederschlag, Verdunstung und Speicherfüllung so groß, dass der Anfangsverlustspeicher überläuft, trifft dieser Input auf die Bodenoberfläche. Wird dabei die Infiltrationskapazität überschritten, entsteht Infiltrations-Überschuss-Abfluss.

Der verbleibende Input wird dem Bodenspeicher hinzugefügt. Erreicht dieser dabei eine Sättigung von mehr als 100%, wird das überschüssige Wasser als Sättigungsüberschuss abgeführt.

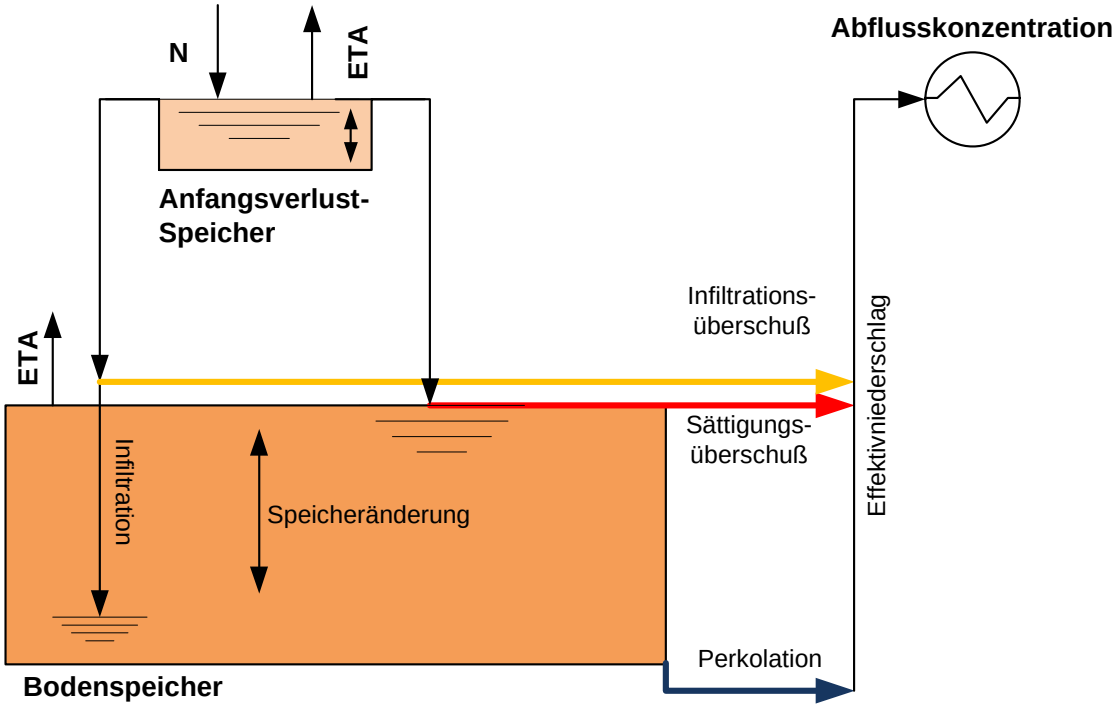


Abb. 4.1 Konzeption der Abflussbildung im ZIN-Modell

Die Sickerung aus der Bodenzelle kann über die ungesättigte Leitfähigkeit $K(\theta)$ nach van Genuchten und dem Darcy-Gesetz berechnet werden (RAWS ET AL.):

$$K(\theta) = K_f \cdot \left(\frac{\theta - \theta_r}{\Phi - \theta}\right)^{\frac{1}{2}} \cdot \left(1 - \left(1 - \left(\frac{\theta - \theta_r}{\Phi - \theta}\right)^{\frac{1}{m}}\right)^m\right)^2 \qquad (4.2)$$

$K(\theta)$: ungesättigte hydraulische Leitfähigkeit

K_f: gesättigte hydraulische Leitfähigkeit

θ: Bodenwassergehalt

Φ: effektive Porosität

mit *m* aus

$$m = \frac{\lambda}{\lambda + 1} \qquad (4.3)$$

λ: Korngrößen-Verteilungsindex nach Brooks and Corey

Der Fluss entspricht nach Darcy dem Produkt aus der hydraulischen Leitfähigkeit und den auftretenden (in Flussrichtung negativen) Gradienten:

$$q = -K(\theta) \cdot \left(\frac{\partial h}{\partial z} + \frac{\partial \psi}{\partial z} \right) \qquad (4.4)$$

$\frac{\partial h}{\partial z}$ *: Höhengradient*

$\frac{\partial \psi}{\partial z}$ *: Matrixpotenzialgradient*

Der Gradient setzt sich zusammen aus der Veränderung des Matrixpotenzials in *z*-Richtung und der Höhenänderung in *z*-Richtung. Der Höhengradient beim Fluss in vertikaler Richtung ist gerade -1. In der jetzigen Modellversion wird angenommen, dass aufgrund der Vernachlässigung von Bodenfeuchteunterschieden innerhalb einer Zelle ein Matrixpotenzialgradient von 0 verwendet werden kann. Dann vereinfacht sich Gleichung 4.4 zu

$$q = K(\theta) \qquad (4.5)$$

Die so berechnete Sickerung kann entweder als Perkolation interpretiert werden, die das System verlässt, oder als Basis- bzw. Zwischenabfluss, der zum Gerinneabfluss beiträgt.

Die Speicheränderung ergibt sich aus der Differenz der Füllungen von Boden- und Anfangsverlustspeicher am Anfang und am Ende der Berechnung.

4.1.4 Abflusskonzentration

In bisherigen Modellversionen wurde mit unterschiedlichen Ansätzen zur Darstellung der Abflusskonzentration gearbeitet. Zum Teil wurden viele Gerinnesegmente in Verbindung mit sehr vielen, kleinen Teileinzugsgebieten verwendet. Dadurch reichen die im Modell dargestellten Wasserläufe sehr weit in die oberen Lagen eines Einzugsgebietes, und der Abstand, der in einem solchen kleinen Teileinzugsgebiet bis zum Gerinne überwunden werden muss, ist sehr klein, so dass die Bedeutung der Abflusskonzentration zugunsten des Routing abnimmt. Für diese Fälle wurde der Niederschlagsinput zum Teil lediglich mit einer zeitlichen Verzögerung dem Gerinne zugefügt (LANGE ET AL. 2000). In anderen Anwendungen wurde eine in einem Teileinzugsgebiet gemessene Konzentrationskurve für die Abflusskonzentration aller Gebiete verwendet.

Alternativ dazu kann auch ein synthetischer Unit-Hydrograph Ansatz verwendet werden. Dieser wurde im Zuge dieser Arbeit weiterentwickelt und wird in Abschnitt 5.4 beschrieben.

4.1.5 Channel-Routing

4.1.5.1 Beschreibung des Muskingum-Verfahrens

Das Muskingum-Verfahren ist ein verbreitetes hydrologisches Verfahren zur Berechnung des Gerinneabflusses. Eine ausführliche Herleitung des Verfahrens findet sich z. B. bei TODINI (2007). Es beruht auf der Massenbilanzgleichung in der Form

$$I - Q = \frac{\partial S}{\partial t} \tag{4.6}$$

wobei I der Zufluss, Q der Abfluss und S der Speicherinhalt eines Gerinnesegmentes zum Zeitpunkt t sind. Durch Diskretisierung der Zeitschritte erhält man

$$\frac{I_t + I_{t+1}}{2} - \frac{Q_t + Q_{t+1}}{2} = \frac{S_{t+1} + S_t}{\Delta t} \tag{4.7}$$

unter der Voraussetzung, dass die Änderungen der Flüsse linear sind. Die Speicheränderung pro Zeitschritt entspricht demnach der Differenz der arithmetischen Mittel aus Zu- und Abfluss des Gerinnesegments von einem Zeitschritt zum nächsten.

Die Speicherfüllungen können auch ausgedrückt werden als

$$S_t = K[I_t + (1 - X)Q_t] \tag{4.8}$$

und für den Fluss aus einem Segment ergibt sich

$$Q_{t+1} = C_1 I_{t+1} + C_2 I_t + C_3 Q_t \tag{4.9}$$

Die Faktoren C_1 bis C_3 haben dann die Werte

$$C_1 = \frac{\Delta t - 2KX}{2K(1 - X) + \Delta t} \tag{4.10}$$

$$C_2 = \frac{\Delta t + 2KX}{2K(1 - X) + \Delta t} \tag{4.11}$$

$$C_3 = \frac{2K(1 - X) - \Delta t}{2K(1 - X) + \Delta t} \tag{4.12}$$

Die Summe der Faktoren C_1 bis C_3 ist 1, so dass diese als Gewichtungsfaktoren der in die Berechnung einbezogenen Abflüsse dienen. Die Tatsache, dass insbesondere der Faktor C_1 dabei negativ werden kann, scheint der Intuition zu widersprechen, tut der Anwendbarkeit des Verfahrens jedoch keinen Abbruch, wie z. B. SZÉL & CSABA (2000) sowie SZILAGYI (1992) darlegen.

Die Parameter K und X lassen sich aus den Eigenschaften des Gerinnes ableiten, für das Muskingum-Verfahren müssen sie im Gegensatz zum Muskingum-Cunge-Verfahren durch Abflussmessungen bestimmt werden.

4.1.5.2 Beschreibung des Muskingum-Cunge-Verfahrens

Das Muskingum-Cunge-Verfahren unterscheidet sich vom Muskingum-Verfahren dadurch, dass die Parameter *K* und *X* nicht als Messwerte eingehen, sondern statt dessen aus den Gerinneparametern berechnet werden:

$$K = \frac{\Delta x}{v_c} \qquad (4.13)$$

sowie

$$X = 0,5 - \frac{Q_{ref}}{2 S_0 \Delta x} \qquad (4.14)$$

Q_{ref} ist dabei ein Referenzabfluss, er entspricht einer Schätzung des Abflusses als dem Mittelwert von Zufluss zum Zeitpunkt *t* und *t + 1* und dem Abfluss zum Zeitpunkt *t*:

$$Q_{ref} = \frac{I_{t+1} + I_t + Q_t}{3} \qquad (4.15)$$

Beim linearen Muskingum-Cunge-Verfahren werden die Parameter entweder für jeden Gerinne-Abschnitt über alle Zeitschritte konstant gehalten, oder sie werden in jedem Zeitschritt einmal aus den Gleichungen 4.13 bis 4.15 berechnet.

Im ZIN-Modell wird das nicht-lineare Verfahren verwendet, bei dem die Berechnung der Parameter K und X aus dem Abfluss und die des Abflusses wiederum mit Hilfe der Parameter *K* und *X* in jedem Zeitschritt mehrfach wiederholt wird, bis die Änderung des Abflusses von einem Iterationsschritt zum nächsten einen festgelegten Wert unterschreitet. Der Ablauf dieser iterativen Lösung ist in Abbildung 5.8 dargestellt.

4.1.5.3 Beschränkungen des Muskingum-Cunge-Verfahrens

Um mit dem Muskingum-Verfahren gute Ergebnisse zu erzielen, muss für die Länge des Zeitschritts folgende Bedingung erfüllt sein (CHOW 1964):

$$2KX \leq \Delta t \leq K \qquad (4.16)$$

Da die Parameter K und X von der Abflussmenge abhängig sind und sich damit im Laufe eines Modelllaufs beträchtlich ändern können, ist diese Bedingung bei festem Zeitschritt und fester Segmentlänge nur im Mittel erfüllbar und es stellt sich die Frage, ob eine Anpassung von Zeitschritt bzw. Segmentlänge auf die Mittelwerte von K und X oder deren bei maximalem Abfluss auftretenden Minima stattfinden soll.

Weiterhin ist das Verfahren nicht in der Lage, Effekte, die flussaufwärts wirken, also Stau-effekte, wie sie Wehre oder Dämme verursachen, darzustellen. Auch für besonders schnell ansteigende Abflüsse, wie Hochwasserwellen nach dem Bruch eines Dammes, ist das Verfahren nicht geeignet (FREAD 1992).

4.2 Das TRAIN-Modell

Das TRAIN-Modell (MENZEL 1999) ist ein räumlich verteiltes Modell zur Berechnung der aktuellen Evapotranspiration nach Penman-Monteith. Sein modularer Aufbau orientiert sich an den dargestellten Prozessen wie Interzeption, Bodenverdunstung und Transpiration.

Zur Bestimmung von Vegetations- und Bodeneigenschaften gibt es vorgegebene Klassen, aus denen der Anwender wählen kann. Die Vegetationseigenschaften enthalten unter anderem die Parameter Interzeptionspeicherkapazität, Leaf-Area-Index, Bestandeshöhe und die Anzahl von Schichten, für die die Berechnungen einzeln ausgeführt werden. Darüber hinaus ist TRAIN in der Lage, die jahreszeitlichen Variationen in den Vegetationseigenschaften zu berücksichtigen. Dazu wird die fortlaufende Zahl des Tages eines Jahres verwendet.

Die Bodenparameter enthalten unter anderem Werte für die Durchwurzelungstiefe, die Feldkapazität *FK*, den permanenten Welkepunkt *PWP*, die nutzbare Feldkapazität *nFK* und den Sättigungswassergehalt.

Für die Kopplung an das ZIN-Modell werden einzelne Module direkt aus dem ZIN-Modell heraus aufgerufen. In der selbstständig laufenden Version werden am ersten Tag der Simulation Startwerte für den Gebietszustand angenommen, in den folgenden Zeitschritten werden dann die Zustandswerte vom vorherigen Zeitschritt übernommen. Ob diese Werteübergabe auch beim

derzeitigen Entwicklungsstand der gekoppelten Version funktioniert, konnte bis zur Fertigstellung dieser Arbeit nicht eindeutig geklärt werden.

4.3 TRAIN-ZIN – Programmstruktur

Der Einstiegspunkt für das TRAIN-ZIN-Programm ist die `main`-Methode. Diese erhält als Startargument den Pfad zur .ctr-Datei, in der alle für den Modelllauf nötigen Angaben enthalten sind. Die Initialisierung weiterer Objekte und Variablen geschieht vom Konstruktor und der zusätzlichen Methode `setup()` der `Execution`-Klasse aus. Der weitere Ablauf wird von der Methode `Execution::run()` gesteuert. Einen Überblick über diesen Ablauf und die wichtigsten Bestandteile des Modells gibt Abbildung 4.2. Dabei wird auch die zentrale Stellung der Klasse `SoilStorage` deutlich, in der der Bodenspeicher und die Abflussbildung modelliert werden.

Die zeitliche Verarbeitungseinheit folgt dem fest vorgegebenen Zeitschritt des TRAIN-Modells von einem Tag, das heißt Abflussbildung, Abflusskonzentration und Channel-Routing werden nacheinander und jeweils für einen ganzen Tag berechnet.

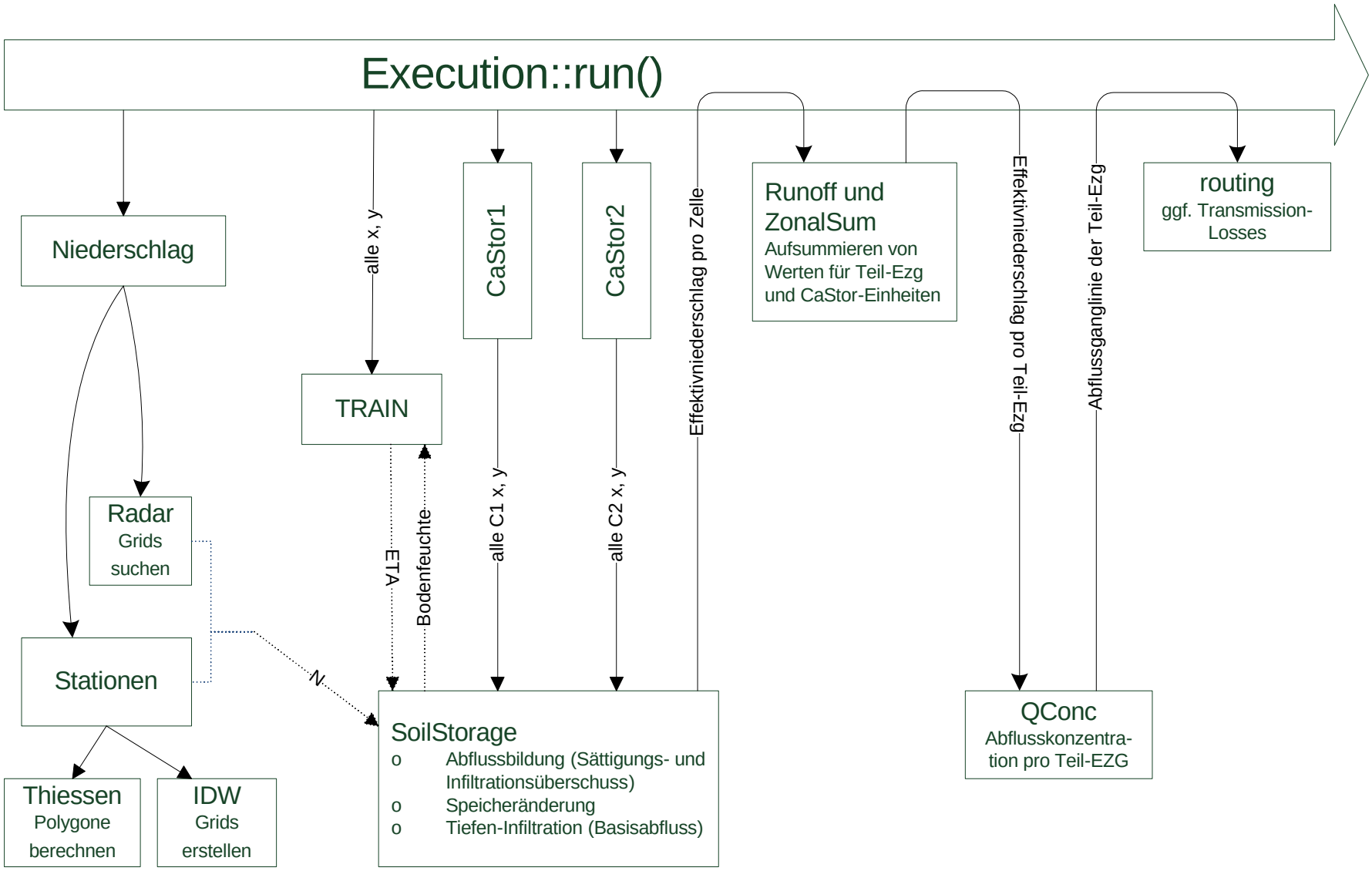


Abb. 4.2 Programmstruktur des TRAIN-ZIN-Modells

Wird das TRAIN-Modell verwendet, werden zuerst für jede Zelle Niederschlags-Tageswerte benötigt. Außerdem übergibt ZIN Bodenfeuchtewerte als Anteil des Wassergehalts zwischen PWP und FK an TRAIN. Für den ersten Zeitschritt werden diese aus der Anfangsfeuchte ermittelt. Anschließend werden nacheinander zuerst die CaStor1 und anschließend (falls vorhanden) die CaStor2-Zellen durchlaufen. CaStor2-Zellen erhalten dabei den Abfluss der CaStor1-Zellen des Teileinzugsgebietes als zusätzlichen Input. Eine zentrale Stellung nimmt die Klasse `SoilStorage` ein. Hier werden aus Niederschlag, Verdunstung sowie für CaStor2-Zellen zusätzlich dem Zufluss aus CaStor1-Zellen, abhängig vom Zustand der Zelle (Speicherfüllung des Bodens und des Anfangsverlustspeichers) und ihren Eigenschaften (konstante Bodenparameter) die Werte für den Oberflächenabfluss (Infiltrations- und Sättigungsüberschuss), Sickerung aus der Bodenzelle heraus nach Van Genuchten sowie die neue Speicherfüllung berechnet. Die in Abbildung 4.2 gestrichelt dargestellte Verbindung zwischen der Klasse `SoilStorage` und dem TRAIN-Modell bzw. der Niederschlagsverarbeitung ist dabei nicht direkt sondern erfolgt über die Methode `Execution::run()`.

Ist die Berechnung der Abflussbildung für einen Tag abgeschlossen, werden aus den Werten der Effektivniederschläge der einzelnen Zellen Summen für jedes Teileinzugsgebiet gebildet. Diese sind die Grundlage für die anschließende Abflusskonzentration, in der der zeitliche Verlauf des Abflusses am Auslass jeden Teileinzugsgebietes berechnet wird. Jedes Teileinzugsgebiet ist genau einem Gerinneabschnitt zugeordnet und liefert so den Input für den letzten Teil des Modells, das Channel-Routing. Dort wird der Wellenablauf, gegebenenfalls unter Berücksichtigung von Transmission-Losses, modelliert.

4.4 Fazit

Das ZIN-Modell ist ein flächenverteiltes, physikalisch basiertes Niederschlags-Abfluss-Modell, dass ursprünglich für den unkalibrierten Einsatz in großen ariden Gebieten konzipiert wurde.

Niederschlagsdaten können dem Modell als Grids aus Niederschlagsradar oder in Form von Stationsdaten zur Verfügung gestellt werden. Die Verteilung von Stationsdaten auf die Fläche erfolgt entweder mit dem Thiessen-Polygon-Verfahren oder dem Inverse-Distance-Weighting-Verfahren.

An das ZIN-Modell ist das ebenfalls flächenverteilte Verdunstungsmodell TRAIN gekoppelt, das basierend auf der Penman-Monteith-Gleichung Tageswerte der Verdunstung berechnet. Die

Verdunstung wird vom ZIN-Modell verwendet, dieses wiederum gibt die Bodenfeuchte am Ende eines Tages an das TRAIN-Modell.

Die Abflussbildung bildet die beiden Oberflächenprozesse des Abflusses aus Infiltrationsüberschuss (Horton-Oberflächenabfluss) und Sättigungsüberschuss sowie die als Basisabfluss interpretierte Perkolation ab.

Die Abflusskonzentration erfolgt nach einem UH-Ansatz, bei dem entweder ein gemessener oder ein synthetischer UH verwendet werden kann.

Das Channel-Routing verwendet das nicht-lineare Muskingum-Cunge-Verfahren.

5 Modellentwicklung

5.1 Das Benutzer-Interface

5.1.1 Anforderungen

Zur Anwendung des TRAIN-ZIN-Modells ist eine Vielzahl von Informationen über das Modell-Setup nötig, die dem Programm in geeigneter Weise zur Verfügung gestellt werden müssen. Diese Informationen werden im Folgenden als Projekt-Informationen bezeichnet, um sie von den wenigen vom Projekt unabhängigen Informationen, die im Quellcode festgelegt sind und den Eingangsdaten (z. B. Messreihen) zu unterscheiden. Zu den Projekt-Informationen gehören

- Pfade zu Ein- und Ausgabedateien
- zu modellierender Zeitraum und Länge von Zeitschritten
- Größe und Anzahl der räumlich verteilten Elemente wie Rasterzellen und Gerinneabschnitte
- hydrologische Parameter
- Auswahl der zu verwendenden Modellkomponenten bzw. Verfahren

In der Ausgangsversion wurden diese Informationen an verschiedenen Stellen direkt im Quellcode eingegeben bzw. die Deaktivierung einzelner Modellkomponenten durch das Auskommentieren von Passagen im Quellcode erreicht. Dieses Verfahren ist zum einen recht unkomfortabel und birgt zum anderen die Gefahr, bei der Veränderung eines Parameters Stellen zu übersehen, die für ein konsistentes Modell-Setup zwingend mit verändert werden müssten. Ein weiterer offensichtlicher Nachteil ist, dass jeder Anwender zumindest über Grundkenntnisse in der Programmierung mit C++ verfügen muss, um mit dem Modell zu arbeiten.

Bezüglich der Projekt-Informationen ergaben sich also folgende Anforderungen:

- Auslagerung aus dem Quellcode
- Zusammenstellung an einer zentralen Stelle
- möglichst unkompliziertes Erstellen eines neuen TRAIN-ZIN-Projekts auch von Anwendern ohne Programmier-Kenntnisse (Benutzer-Interface)
- geeignetes Verfahren zur Einbindung der Projekt-Informationen in das Programm (Programm-Interface)

- Flexibilität auf Benutzer- und Programmseite, die eine Weiterentwicklung des Programms erlaubt, ohne dass die Struktur der Schnittstellen geändert werden müsste

Für einen größtmöglichen Benutzer-Komfort wurde zunächst die Erstellung einer grafischen Benutzeroberfläche (GUI) erwogen. Allerdings unterscheiden sich die Verfahren zur Entwicklung eines GUI wesentlich von denen zur Entwicklung des restlichen Programms. Insbesondere wird in der verwendeten Entwicklungsumgebung MS-Visual-Studio (VS) zur Erstellung von GUIs („Windows-Forms"-Projekte) die Bibliothek C++/CLI verwendet. Diese Unterscheidet sich in ihrer Syntax erheblich von Standard-C++. Dadurch wäre nicht nur das Erstellen, sondern auch das laufende Anpassen der Benutzeroberfläche an ein verändertes Programm ein erheblicher Aufwand, womit die Forderung nach Flexibilität bei der Programmentwicklung verletzt gewesen wäre. Statt dessen wurde als User-Interface eine Text-Datei (ASCII-File) gewählt.

5.1.2 Das ASCII-Interface

Das Interface besteht aus einer Textdatei (ASCII-Zeichen), die zum Erstellen eines TRAIN-ZIN-Projektes verwendet wird. Diese kann vom Anwender in jedem Texteditor bearbeitet werden. Sie wird im folgenden als „Steuerungsdatei" oder nach der Dateinamenerweiterung als „.ctr-Datei" bezeichnet.

Die Steuerungsdatei enthält Paare aus Schlüsseln und Werten. Der Schlüssel besteht aus einem Wort, das vom Programm zur eindeutigen Identifizierung des Wertes benutzt wird. Das Schlüsselwort muss der erste Eintrag in einer Zeile sein (außer Leerzeichen und Tabulatoren). Als Werte-Typen können Ordnernamen, Dateinamen, Zahlen, Wörter und Wahrheitswerte vorkommen. Der zu einem Schlüssel gehörende Werte-Typ ist festgelegt, so dass der Benutzer darauf achten muss, dass sich seine textuelle Eingabe tatsächlich in den verlangten Typ konvertieren lässt. Ist dies nicht möglich, etwa weil statt der erwarteten Zahl ein Buchstabe eingegeben wird, bricht das Programm mit einer entsprechenden Fehlermeldung ab.

Für die verschiedenen Wertetypen sind folgende Eingaben zulässig:

- Ordnernamen: Unterordner werden mit einfachem / oder \ gekennzeichnet, am Ende steht ebenfalls / oder \
  - Projektordner: der Ordner, in dem aller In- und Output liegt. Die Angabe kann als vollständiger Pfad oder als relativer Pfad erfolgen. Bei der Verwendung unter VS bezieht sich eine relative Angabe auf den Ort der ZIN-Projektdatei (ZIN.vcproj),

bei Verwendung aus der Konsole heraus auf den Ort der ausführbaren Datei TrainZin.exe

  - weitere Ordner: Angaben beziehen sich grundsätzlich auf den Projektordner, eine Angabe von absoluten Pfaden ist hier nicht möglich

- Dateinamen können Pfadangaben enthalten, wiederum relativ zum Projektordner. Der Dateiname muss vollständig mit seiner Erweiterung angegeben werden.
- Zahlen: es werden Ganzzahlwerte (`integer`) und Gleitkommatypen (`float` und `double`) unterschieden.
- Wahrheitswerte (`bool`) werden durch die Werte 0 (falsch) und 1 (wahr) dargestellt

Für alle Eingaben gilt, dass sie keine Leerzeichen, Tabulatoren oder Zeilenumbrüche enthalten dürfen, weil diese Zeichen als Ende der Eingabe interpretiert werden.

Werte, die in einem Setup nicht benötigt werden, müssen nicht spezifiziert werden.

Kommentare sollten mit einem %-Zeichen gekennzeichnet werden. Dies ist für die Funktion nicht zwingend notwendig, dient aber der Klarheit und beschleunigt die Verarbeitung etwas.

5.1.3 Die Programm-Seite des Interface

Zur Interaktion zwischen Programm und ASCII-Datei wurde eine Klasse „`Controller`" angelegt. Für den Ablauf des Programms ist genau eine Instanz dieser Klasse nötig, die bereits in der `main`-Methode angelegt wird. Die *Adresse* dieses Objekts wird an alle anderen Objekte weitergegeben. Der Konstruktor führt folgende Schritte aus:

- Öffnen der Steuerungsdatei (ggf. Fehlermeldung, wenn dies misslingt)
- Überprüfen, ob das erste Wort in der Datei „ctr" ist, um sicherzustellen, dass es sich um eine für diesen Zweck erstellte Steuerungs-Datei handelt
- Überlesen des Inhalts bis zum Schlüsselwort „startcoding"
- Einlesen des folgenden Textes als `string`, wobei Zeilen ab dem Kommentar-Zeichen „%" ignoriert werden
- Initialisierung eines Input-Streams mit dem gelesenen Inhalt

Der Input-Stream enthält nun abwechselnd einen Schlüssel und einen Wert, und das `Controller`-Objekt ist im Arbeitszustand.

Die Beziehungen der Methoden der `Controller`-Klasse untereinander und mit der aufrufenden Seite sind in Abbildung 5.1 dargestellt. Dabei ist zwischen öffentlichen (`public`) und privaten (`private`) Methoden unterschieden. Als `public` deklarierte Klassenelemente sind diejenigen Elemente, auf die von außerhalb der Klasse zugegriffen werden kann. Ihre interne Funktionsweise ist außerhalb der Klasse nicht sichtbar und spielt für die Verwendung auch keine Rolle. Dieses für eine objektorientierte Programmiersprache wesentliche Merkmal wird als Kapselung bezeichnet (BREYMANN 1999). Auf private Methoden kann nicht von außerhalb der Klasse zugegriffen werden.

Die privaten Methoden haben folgende Funktionen:

- `getElement(string)`: sucht nach dem Schlüssel und liefert das darauf folgende Element als `string` zurück
- `pathExists(string)`: erkennt, ob ein Pfad oder ein Dateiname übergeben wurde. Im günstigen Fall wird `true` zurückgegeben, `false` sonst
  - Pfade: überprüft, ob der Pfad existiert, indem versucht wird, eine Datei in den Ordner zu schreiben (die Datei wird sofort wieder gelöscht)
  - Datei zum Lesen: versucht, die Datei zum Lesen zu öffnen
  - Datei zum Schreiben: versucht, die Datei zum Schreiben zu öffnen (der Inhalt wird dabei nicht überschrieben)
- `getElem(string)` ist als `Template` implementiert. Damit ist sie typunabhängig und kann jeden integralen Typen als Rückgabewert haben. Sie verwendet die Methode `getElement`, um den gesuchten Wert als `string` zu erhalten. Dann wird versucht, diesen string in den gewünschten Typen umzuwandeln und zurückzugeben. Gelingt dies nicht, wird eine Fehlermeldung ausgegeben und der Anwender aufgefordert, einen passenden Wert einzugeben oder das Programm durch Eingabe von ‘q’ zu beenden.
- `getNewFilename(string)` fordert den Benutzer so lange zur Eingabe eines neuen Pfad- oder Dateinamens auf, bis dieser gültig ist oder der Benutzer das Programm durch die Eingabe von ‘q’ beendet.

Die öffentlichen Methoden haben die folgenden Funktionen:

- `getPath(string)` verwendet die M ethode `getElement`, um einen Datei- oder Pfadnamen zu extrahieren. Dieser wird mit dem Pfad des Projektordners verkettet und an

die Methode checkPath übergeben, um dessen Gültigkeit sicherzustellen. Anschließend wird der (ggf. neue) Pfad- oder Dateinamen zurückgegeben.

- `checkPath(string)` verwendet die Methode `pathExists` um die Gültigkeit eines Pfad- oder Dateinamens zu überprüfen. Ist dieser ungültig, wird ein von der Methode `getNewFilename` beschaffter neuer Dateiname zurückgegeben, ansonsten der alte.
- `checkWrite(string)` fordert den Benutzer auf, einen neuen Pfad- oder Dateinamen einzugeben, solange der Pfad- oder Dateiname ungültig ist (Überprüfung mit `pathExists`) oder bis der Benutzer das Programm durch die Eingabe ‘q’ beendet.
- `getBool`, `getInt`, `getFloat`, und `getString` rufen die jeweilige Instanz der Methoden-`Template getElem` auf und geben den erhaltenen Wert zurück.

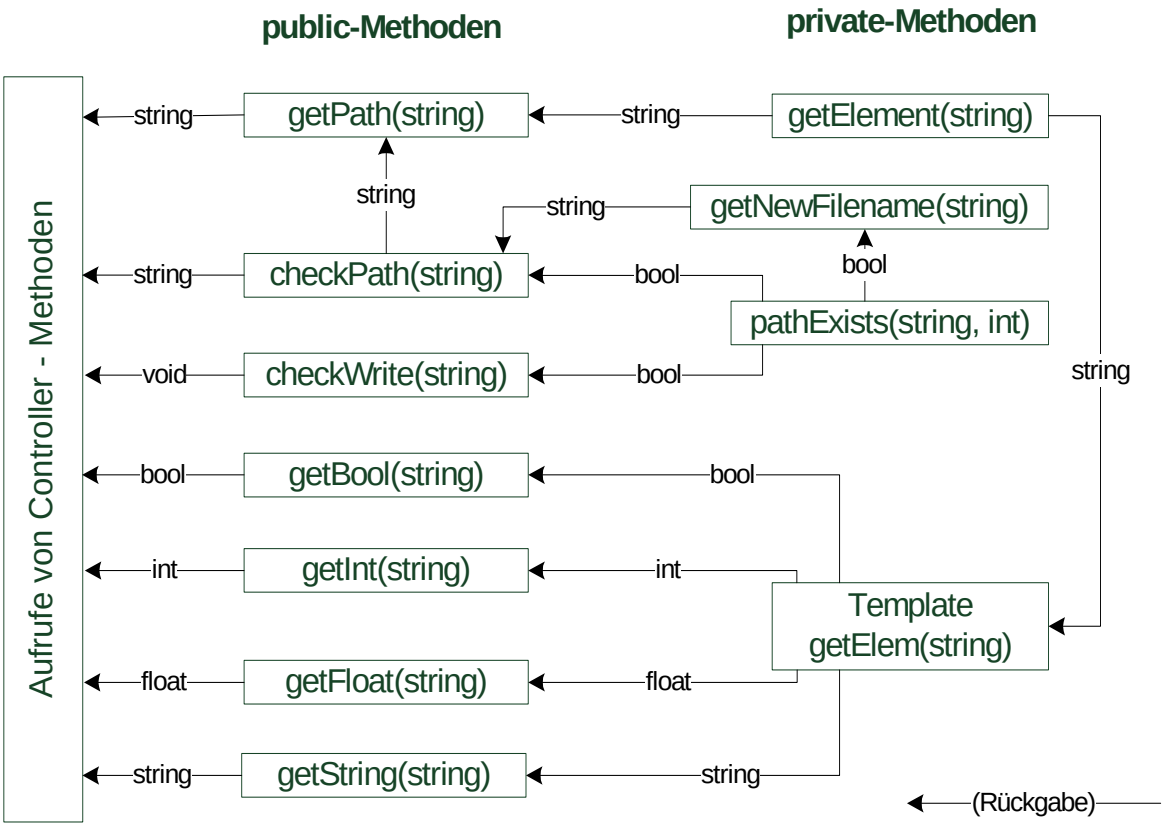


Abb. 5.1 Struktur der `Controller`-Klasse (Aufruf in umgekehrter Richtung der Rückgabe)

5.1.4 Beschränkungen

Eine wesentliche Einschränkung beim Einsatz der `Controller`-Klasse im Programm liegt in der Laufzeit. Versuche ergaben, dass ein Aufruf einer der get-Methoden in jedem ZIN-Zeitschritt für jede Rasterzelle das Programm bis zur Unbrauchbarkeit verlangsamt. Für Elemente, die deutlich mehr als einmal pro Zeitschritt benötigt werden (z. B. die Frage, ob der TRAIN-Teil verwendet werden soll), sollte deshalb eine Instanzvariable angelegt werden, in der der zugehörige Wert abgelegt wird und auf die bedeutend schneller zugegriffen werden kann.

5.2 Verarbeitung von Stationsdaten für Niederschlag

Bislang war TRAIN-ZIN nur in der Lage, entweder Rasterdaten (gewöhnlich aus Radardaten) in Fünf-Minuten-Auflösung oder räumlich homogenen Niederschlag zu verarbeiten. Um Stationsdaten zu verwenden, musste die Berechnung des Gebietsniederschlags außerhalb des Modells als Teil der Datenaufbereitung z. B. mit Hilfe eines GIS erfolgen.

Der dafür jeweils notwendige Zeitaufwand und die Größe der Datenmenge ließen eine Verarbeitung innerhalb der Modell-Software erstrebenswert erscheinen. So betrug die Größe eines Niederschlagsgrids für das Dragonja-Gebiet (ca. 370.000 Zellen) im Mittel etwa 2,5 Megabyte. Um Niederschlagsgrids dieser Größe für die Simulation eines Jahres in Fünf-Minuten-Zeitschritten anzulegen, bräuchte man also ca. 257 Gigabyte Festplattenspeicher. Dies überstieg die zur Verfügung stehenden Speicherkapazitäten.

Wesentliche Aspekte für die Integration eines stationsbasierten Niederschlagsmoduls waren eine räumliche Verteilung der Stationsdaten nach bewährten Verfahren, ohne dabei die Struktur auf ein einzelnes Verfahren festzuschreiben, um eventuelle spätere Erweiterungen nicht zu erschweren.

Für die räumliche Verteilung der Stationsdaten wurden zwei Verfahren implementiert: das Thiessen-Polygon- und das Inverse-Distance-Weighting-Verfahren (IDW). Beim IDW ist optional eine DEM-basierte Höhenkorrektur möglich.

5.2.1 Thiessen-Polygone

Die Aufteilung eines Gebietes in Polygone nach Thiessen ist ein einfaches Verfahren, um jedem Punkt (bzw. jeder Rasterzelle) im Gebiet den Wert einer Niederschlagsstation zuzuweisen. Die Polygone sind einfach grafisch zu konstruieren und eignen sich daher gut für die

manuelle Anwendung ohne Hilfe von Computern. Schwächen des Verfahrens sind die mangelnde Berücksichtigung orografischer Effekte und schlechte Repräsentation der räumlichen Verteilung bei großer räumlicher Niederschlagsvariabilität. (MANIAK 2005)

Von Vorteil bei der rechnergestützten Verarbeitung ist der sehr geringe Aufwand an Rechenzeit und Speicherplatz. Außerdem bietet sich die Möglichkeit, räumlich homogenen Niederschlag für ein ganzes Gebiet anzunehmen.

Bei der Konstruktion der Polygone wird das Gebiet derart unterteilt, dass genau diejenigen Zellen zu einem Polygon gehören, deren Abstand zur zugehörigen Station geringer ist, als zu allen andern Stationen (CROLEY ET AL. 1985). Um die Einteilung vornehmen zu können, wird also für jede Zelle ihre Entfernung zu jeder Station benötigt. Um diese Berechnung nicht unnötig wiederholen zu müssen, wird im Programm ein Grid-Array in der Größe der Zahl von Stationen angelegt (Variablenname im Quellcode `distGrid`), sodass für jede Station ein Grid existiert, das an jeder Stelle (*x, y*) die Entfernung zu der betreffenden Station enthält.

Der nächste Schritt ist die Erstellung eines Grids, das an jeder Stelle (*x, y*) den Index der nächstliegenden Station enthält. Mit Hilfe dieses Index kann dann bei der Abfrage des Niederschlagswertes unmittelbar auf die richtige Station zugegriffen werden. Dazu wird für jede Zelle ein Integer-Array (im Programm: `distorder`) verwendet, das alle Stationsindizes in anfänglich beliebiger Reihenfolge enthält. Die Indizes in diesem Array werden dann derart sortiert, dass an der ersten Stelle der Index der nächstliegenden Station steht, an der zweiten Stelle der Index der zweitnächsten Station usw. Tabelle 5.1 gibt beispielhaft den Inhalt von `distorder` nach dem Sortieren für ein Setup mit vier Stationen wieder.

Tabelle 5.1 Beispiel für die Sortierung der Stationsindizes nach Entfernung zwischen Zelle und Station

Index i	Wert von `distorder` an der Stelle i (Stationsnummer)	Entfernung (Sortierschlüssel)
0	1	12,3
1	3	30,7
2	0	100,4
3	2	540

Als Sortieralgorithmus wird „Bubblesort” verwendet, da es sehr effizient für gut vorsortierte Folgen ist, wie dies bei benachbarten Zellen meist der Fall ist.

Diese vollständige Sortierung wäre nicht nötig, um nur ein Grid mit den Indizes der nächstliegenden Station zu erstellen. Ist allerdings für einen Zeitschritt an einer Station kein Messwert vorhanden, kann durch das Speichern aller Indizes auf im Bezug auf die Entfernung nachfolgende Stationen zugegriffen werden. Deshalb wird ein Grid-Array `stationGrid` verwendet mit einem Grid für jede Niederschlagsstation. Dies ist die zweidimensionale Entsprechung der Variablen `distorder`.

Das Abrufen des Niederschlagswertes für eine Zelle ist so implementiert, dass `stationGrid` mit aufsteigendem Index durchsucht wird, bis ein gültiger Niederschlagswert gefunden wurde. In den meisten Fällen wird dies schon bei der nächsten Station (beim Index 0) der Fall sein. Alternativ kann in der Steuerungsdatei eingestellt werden, dass „noData” als 0 interpretiert wird, ohne in weiter entfernten Stationen Werte abzurufen.

5.2.2 Inverse-Distance-Weighting

Das Inverse-Distance-Weighting-Verfahren ist ein weitverbreitetes Verfahren zur Berechnung des Gebietsniederschlags aus Stationsdaten (SMITH 1992). Für das ZIN-Modell wurde es auch bisher schon verwendet (THORMÄHLEN 2003), indem die Grids außerhalb des Modells erzeugt wurden.

Der Niederschlagswert für eine Rasterzelle berechnet sich aus einem gewichteten Mittel einer Anzahl von Stationswerten P_i. Die Wichtung erfolgt mit dem Quadrat des Abstandes d zwischen der Station i und der Rasterzelle j. Der Niederschlag $\overline{P}_j$ der Zelle j ist dann

$$\overline{P}_j = a \sum_{i=1}^{n} \frac{1}{d_{ij}^2} P_i \qquad (5.1)$$

mit

$$a = \left(\sum_{i=1}^{n} \frac{1}{d_{ij}^2} \right)^{-1} \qquad (5.2)$$

Der Exponent *2* für die Gewichtung des Abstandes zeigt dabei nach empirischen Tests von SHEPARD (1968) die besten Ergebnisse. Größere Exponenten führen demnach zu flachen Regionen um die gemessenen Werte herum mit steilen Übergängen zur nächsten Station,

während kleinere Exponenten eine insgesamt flache Fläche mit einzelnen Erhebungen um die Stationen herum führen.

Ein Schwachpunkt des Verfahrens ist, dass nur Entfernungen, nicht aber die Richtungen zu einer Niederschlagsstation D_i berücksichtigt werden. Die beiden in Abbildung 5.2 dargestellten Konfigurationen führen daher zu dem gleichen Ergebnis für den Niederschlag am Punkt *P*, obwohl man erwarten würde, dass im Fall *(a)* der Einfluss der Station D_1 von D_2 abgeschirmt wird und im Fall *(b)* der Einfluss der Station D_3 von der Station D_2.

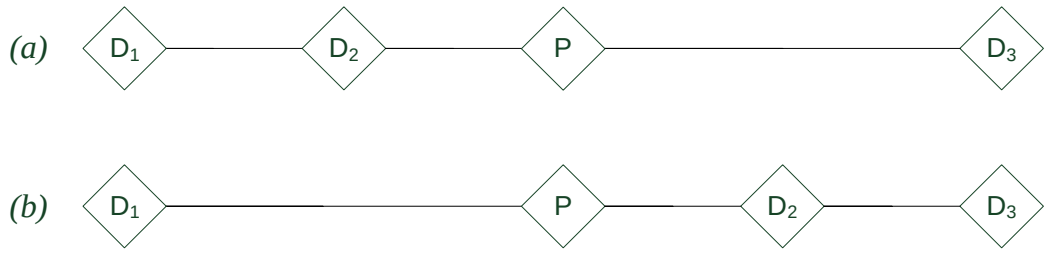


Abb. 5.2 Zwei Anordnungen mit gleichem IDW-Ergebnis für Punkt *P* (nach SHEPARD 1968)

Außerdem werden Ansammlungen von mehreren Stationen (Cluster) je nach Lage von Stationen und Punkten übermäßig stark gewichtet. Abbildung 5.3 zeigt eine Anordnung, bei der der Datenpunkt *P* die gleiche Entfernung *a* zu allen Stationen hat. Das IDW-Verfahren wird dazu führen, dass *P* das arithmetische Mittel der Stationen zugewiesen wird, obwohl die dicht beieinanderliegenden Stationen D_1 und D_2 offensichtlich je ein geringeres Gewicht erhalten sollten als die einzelne Station D_3.

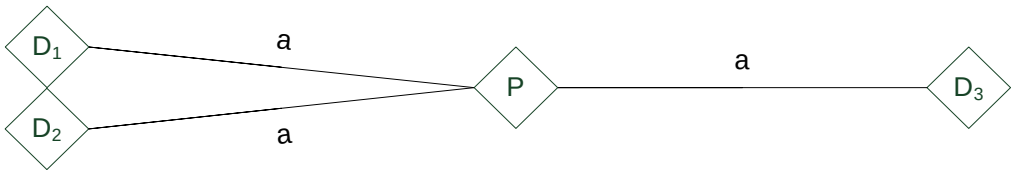


Abb. 5.3 Übermäßige Gewichtung von Stations-Clustern mit dem IDW-Verfahren

5.2.2.1 Implementation im TRAIN-ZIN-Modell

Da das Gewicht einer Station mit der Entfernung stark abnimmt, ist es zweckmäßig, die Zahl der verwendeten Stationen zu begrenzen, da weitere Stationen keinen Zugewinn an Genauigkeit bringen, den Rechenaufwand aber erhöhen. Eine Möglichkeit besteht darin, nur Stationen innerhalb eines festgelegten Radius zu verwenden. Sind die Stationsdaten allerdings lückenhaft, muss der Radius entweder von vornherein so groß gewählt werden, dass für jede Zelle immer mindestens eine Station mit Daten erreicht wird, oder der Radius muss sich bei Bedarf flexibel erweitern lassen.

In der vorliegenden Implementation wird stattdessen eine festgelegte Anzahl von Stationen verwendet. Die Zahl der zu verwendenden Stationen wird in der Steuerungsdatei festgelegt, für die Modellierung des Dragonja-Gebietes wurden jeweils drei Stationen verwendet.

Gibt es weniger Stationen, die Daten enthalten, als der Anwender für die Verwendung vorgesehen hat, werden stattdessen nur die tatsächlich vorhandenen Stationen verwendet. Dies funktioniert sowohl für Stationen, die das gewünschte Datum gar nicht enthalten, als auch für solche, die nur für den betreffenden Zeitschritt keine Daten enthalten (Datum mit „noData").

Liegen die Stationen auf unterschiedlichen Höhen, sodass der Niederschlag von der Lage beeinflusst ist, ist die Anwendung des IDW-Verfahrens nicht unmittelbar möglich. Daher wurde die optionale Verwendung einer Höhenkorrektur des Niederschlags implementiert. Dazu werden die Niederschlagswerte zuerst mit Hilfe eines anzugebenden relativen Gradienten (%/100 m) auf ein Bezugsniveau umgerechnet:

$$P_{ref} = \frac{P_{Stat}}{1 - grad \cdot \left(h_{Stat} - h_{ref} \right)} \tag{5.3}$$

Nach dem Durchlaufen des IDW werden die Werte der einzelnen Zellen dann unter Verwendung des gleichen Gradienten auf das Niveau der Zelle umgerechnet:

$$P_j = P_{ref} \cdot \left(1 + grad \cdot \left(h_j - h_{ref} \right) \right) \tag{5.4}$$

Als Beispiel ist in Abbildung 5.4 die mit dem Modell erzeugte Niederschlagskarte der Tagessumme für den 11.5.2002 dargestellt. Um die Stationen Marezige und Kojcancici sind die für das IDW-Verfahren typischen kreisförmigen Strukturen zu erkennen. Die entlang einer Linie verlaufenden Werte-Sprünge kommen durch die Verwendung einer begrenzten Anzahl von

Stationen zustande und markieren den Wechsel von einer der drei für jede Zelle verwendeten Stationen.

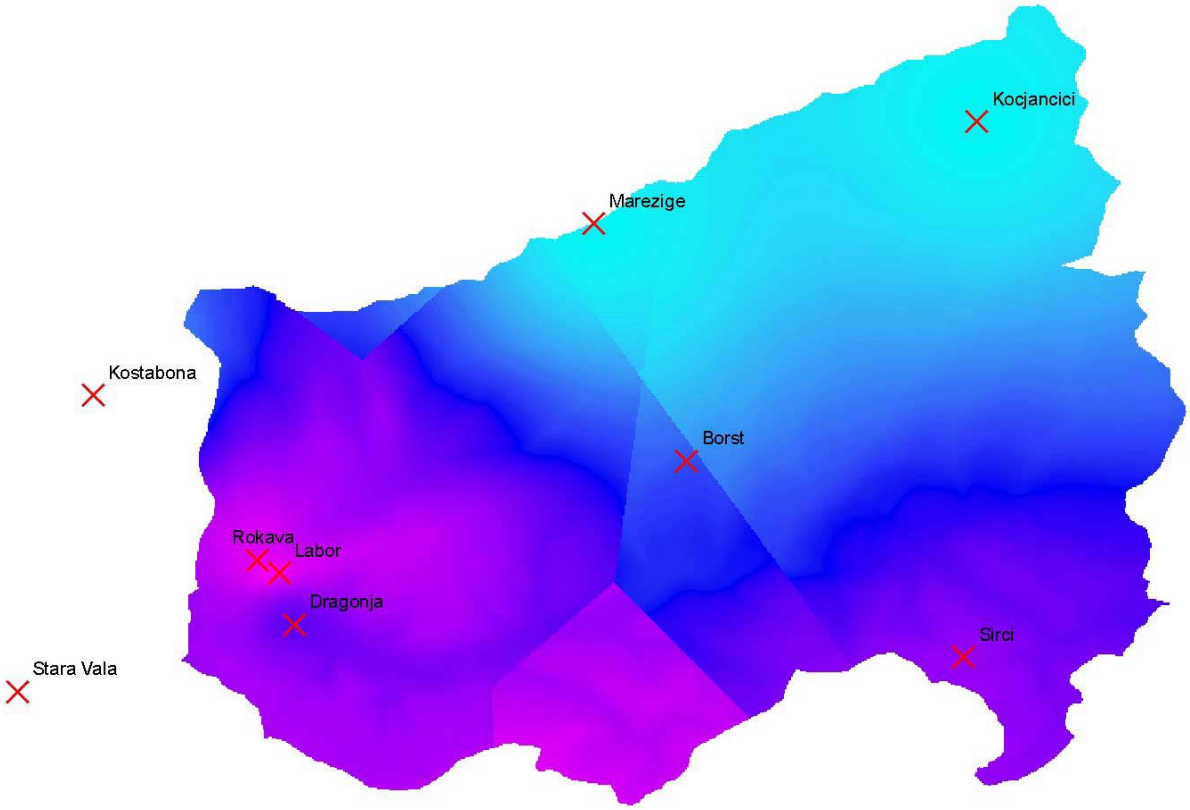


Abb. 5.4 Inverse-Distance-Grid für den 11.5.2002 (Tagessumme, Werte 0 bis 18 mm, Station Labor ohne Daten)

5.3 Abflusskonzentration

5.3.1 Bisherige Situation

Die Abflusskonzentration im TRAIN-ZIN-Modell erfolgt nach dem Unit-Hydrograph-Konzept (UH) von SHERMAN (1932) für die einzelnen Teileinzugsgebiete. Eine Herausforderung besteht dabei darin, für jedes Teileinzugsgebiet eine passende Übertragungskurve zu finden.

In der von Lange 1999 erstellten Version des ZIN-Modells wurde eine Abflusskonzentrationskurve für ein Teileinzugsgebiet durch Messungen im Gelände ermittelt und angenommen, dass sie auf alle anderen Teileinzugsgebiete anwendbar ist.

Voraussetzung hierfür ist eine ausreichende Ähnlichkeit der Teileinzugsgebiete. Unterscheiden sie sich jedoch hinsichtlich der Größe, der Form und der Hangneigung, kann man

davon ausgehen, dass sich die Antwortfunktion deutlich unterscheidet. Außerdem muss die Möglichkeit gegeben sein, eine solche Kurve vor Ort zu erheben.

Um diesem Problem zu begegnen, implementierte FISCHER 2007 die Verwendung eines synthetischen UH, bei der die Größe des Teil-Einzugsgebietes sowie seine Zugehörigkeit zu einer von drei Hangneigungsklassen eingingen.

5.3.2 Weiterentwicklung des UH-Ansatzes

Da für die Modellierung des Dragonja-Gebietes keine gemessene Abflusskonzentrationskurve vorlag oder zu beschaffen war, wurde der Ansatz von Fischer weiterverfolgt.

Ziele bei der Weiterentwicklung waren

1. Auswahl einer Verteilung, die die erwartete Form einer Abflussganglinie besser wiedergibt
2. direkte Verwendung der Hangneigung ohne Einteilung in Klassen
3. die Verwendung einer gemessenen Konzentrationskurve sollte weiterhin möglich sein
4. Verbesserung der Sicherheit der Implementierung bezüglich des Umgangs mit Arrays und dem möglichen Zugriff auf undefinierte Speicherbereiche.

Eine häufig zur Erstellung eines synthetischen Unit-Hydrographs verwendete Verteilung ist die Gamma-Verteilung, da sie das Verhalten der häufig als Modellkonzept verwendeten seriellen Einzellinearspeichern wiedergibt (MANIAK 2005):

$$f(t) = \frac{t^{\alpha-1}}{\beta^{\alpha}\Gamma(\alpha)} \cdot e^{-\frac{t}{\beta}} \qquad (5.5)$$

$\Gamma(\alpha)$ ist dabei die Gammafunktion, die der Gamma-Verteilung ihren Namen gibt. Für ganze Zahlen entspricht $\Gamma(\alpha)$ der Fakultät von α, für reelle Zahlen ist sie die Erweiterung der Fakultätsfunktion auf die reellen Zahlen. α und β sind Parameter, die den Kurvenverlauf bestimmen. Bei gegebener Konzentrationszeit stehen die beiden Parameter in folgendem Verhältnis (NADARAJAH 2007):

$$\beta = \frac{t_{konz}}{\alpha - 1} \qquad (5.6)$$

t_{konz}: Konzentrationszeit

Croley (1980) weist darauf hin, dass eine Verwendung der Gamma-Verteilung als dimensionslosem UH zur Übertragung zwischen Einzugsgebieten nicht sinnvoll ist, da die geeigneten Parameter sich zwischen unterschiedlichen Einzugsgebieten sowie zwischen den Niederschlagsereignissen zu stark unterscheiden.

Abbildung 5.5 zeigt die Gamma-Verteilung für verschiedene Parameter-Paare (α, β) nach Gleichung 5.6 mit einer konstanten Konzentrationszeit von 30 (dimensionslos). Die Kurven unterscheiden sich offensichtlich nicht nur bezüglich ihrer Höhe, sondern auch in ihrer Form, sind also nicht durch eine Skalierung ineinander zu überführen.

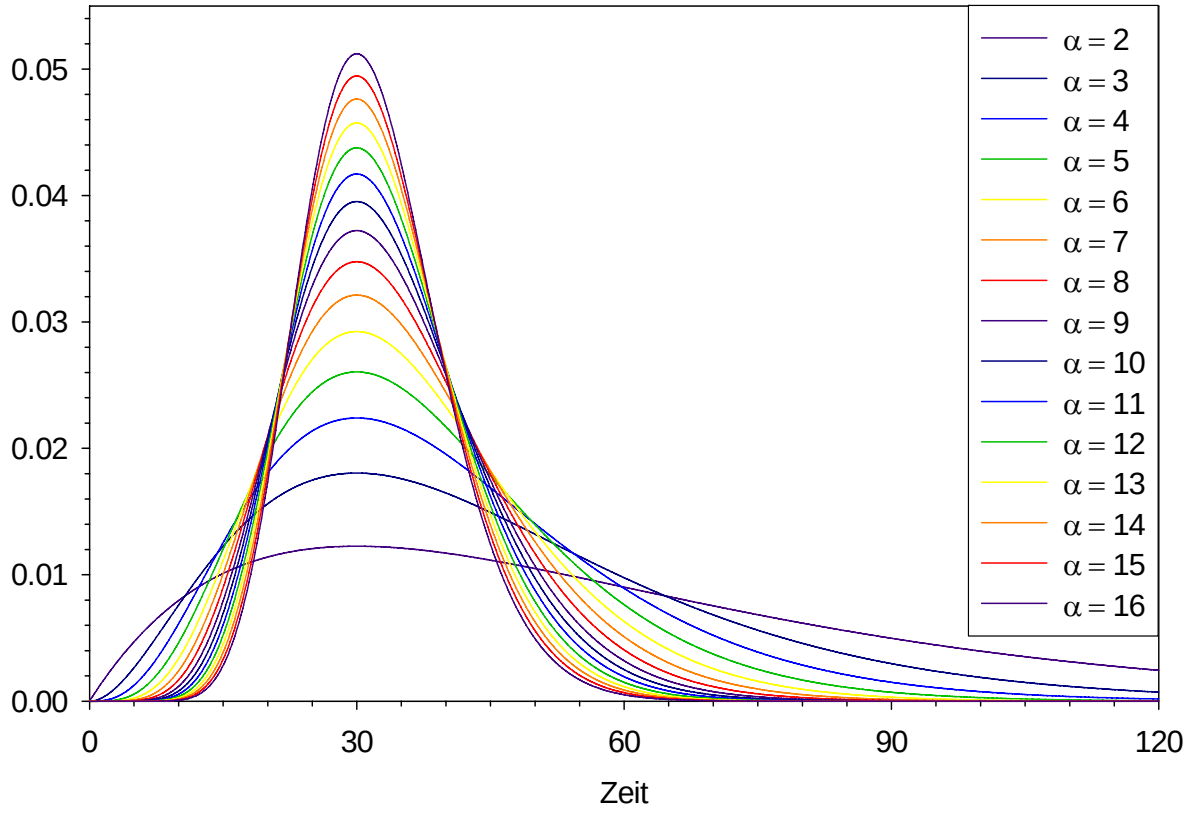


Abb. 5.5 Gamma-Verteilungen mit einer Konzentrationszeit von 30 bei verschiedenen Parameter-Paaren (α, β), Berechnung von β nach Gleichung 5.7

Die Auswahl geeigneter Parameter wäre bei den beschränkten zur Verfügung stehenden Informationen über die Teileinzugsgebiete im Dragonja-Gebiet also mit großer Unsicherheit behaftet gewesen.

Statt der Gamma-Verteilung wurde daher die Extremwertverteilung Typ I (EV I) gewählt (Gleichung 5.7). Andere Bezeichnungen für diese Verteilungsfunktion sind Gumbel-Verteilung oder Fisher-Tippett-Verteilung für Extremwertverteilungen allgemein (RAO ET AL. 2000):

$$f(t) = \frac{1}{b} e^{-\frac{t-a}{b}} \exp\left(-e^{-\frac{t-a}{b}}\right) \qquad (5.7)$$

Der Abflussbeitrag *q* zum Gesamtabfluss eines Modellierungszeitschrittes ergibt sich dann aus

$$q(t) = f(t) \cdot V \qquad (5.8)$$

mit *V* als dem zum Zeitschritt *t = 0* gebildeten Abflussvolumen.

Die EV I ist mit lediglich zwei Parametern sehr einfach. Der Parameter *a* entspricht der Konzentrationszeit und bewirkt eine Verschiebung der gesamten Kurve auf der Zeitachse ohne Veränderung der Form. Der Parameter *b* ist ein Skalierungsfaktor, der ebenfalls nicht die Form verändert, sondern sie lediglich staucht bzw. auseinanderzieht.

Um diese Verteilung nutzen zu können, müssen die Bedingungen

$$q(t) = 0, \quad t < 0 \qquad (5.9)$$

und

$$\int_0^{t_{max}} q(t)\,dt = V \qquad (5.10)$$

erfüllt sein. Gibt es Abfluss für *t < 0*, ergibt sich damit auch eine Verletzung der Bedingung aus Gleichung 5.10. Diese wird außerdem dadurch verletzt, dass nicht ein unbegrenzt langes Tailing berücksichtigt werden kann. Für das Auslaufen werden Werte bis zum Erwartungswert zuzüglich des achtfachen der Standardabweichung berücksichtigt:

$$t_{max} = E(t) + 8 \cdot \sigma = a + b\gamma + 8 \cdot \frac{\pi b}{\sqrt{6}} \qquad (5.11)$$

wobei γ die Euler-Mascheroni-Konstante bezeichnet. Bei der Berechnung einer Kurve werden die Funktionswerte jeden Zeitschritts aufsummiert. Anschließend wird jeder Funktionswert durch diese Summe geteilt, sodass die neue Summe danach wieder eins ergibt und die Bedingung aus Gleichung 5.10 erfüllt ist.

Diese Korrektur ist nun auch bei der Variante mit gemessener Abflusskurve möglich. Hier wird eine Warnung ausgegeben, wenn die Summe der Einzelwerte um mehr als 0,01 von eins abweicht und der Anwender aufgefordert, zwischen einem Abbruch des Programms und der Korrektur der Werte zu wählen.

Um die Variabilität des UH zwischen den Einzugsgebieten zu berücksichtigen, wurde vereinfachend ein Zusammenhang zwischen der Konzentrationszeit (Parameter *a*) und der Hangneigung sowie zwischen der zeitlichen Ausdehnung des UH (Parameter *b*) und der Teileinzugsgebietsfläche angenommen. Wirkungen der Fläche auf die Konzentrationszeit, der Hangneigung auf die Breite, der Einfluss der Einzugsgebietsform sowie weitere Einflüsse wurden vernachlässigt.

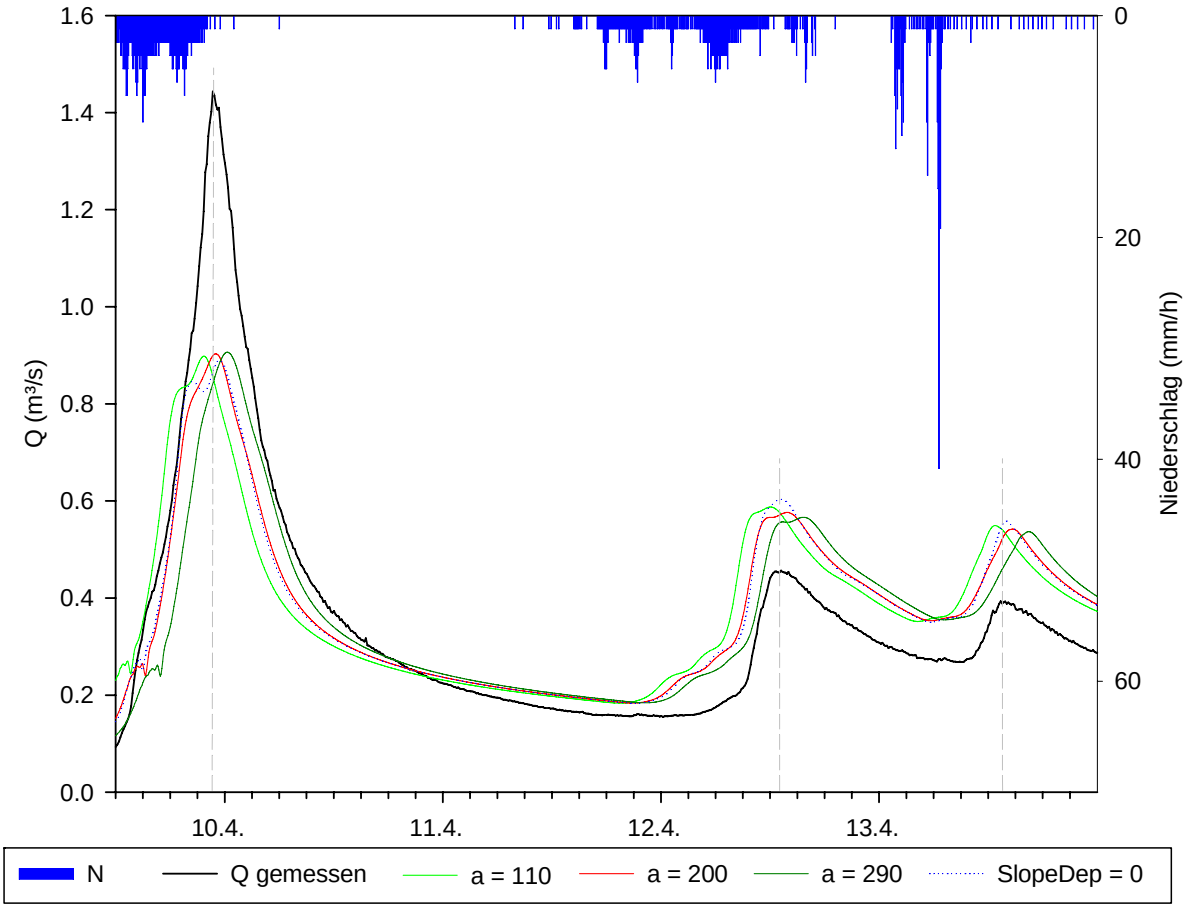


Abb. 5.6 Einfluss des Abfluss-Konzentrations-Parameters *a*

Abbildung 5.6 zeigt den Einfluss des Parameters *a* anhand einer Simulationsperiode im April 2004. Dabei wurde *a* um einen Standardwert von 200 Minuten um 90 Minuten nach oben und nach unten verändert. Die Verschiebung um je 90 Minuten spiegelt sich unmittelbar in der Ganglinie am Gebietsauslass wieder, die Veränderung der Form ist dabei gering. Eine Simulation ohne eine Abhängigkeit des Parameters *a* von der Hangneigung (SlopeDep = 0) führte lediglich zu einer geringfügigen Änderung der Form der Ganglinie.

Abbildung 5.7 zeigt den Einfluss des Parameters *b* für dieselbe Simulationsperiode. *b* wurde gegenüber dem Standardwert von 40 halbiert bzw. verdoppelt. Für *b* = 20 verläuft die Kurve um den Scheitelpunkt eines Abflussereignisses etwas steiler, Schwankungen im Niederschlag führen eher zu mehrfachen Abflusspeaks (Simulation am 10.4.02 und 12.4.02). Für *b* = 80 verläuft der Abfluss etwas flacher und dicht aufeinander folgende Peaks können verschmelzen.

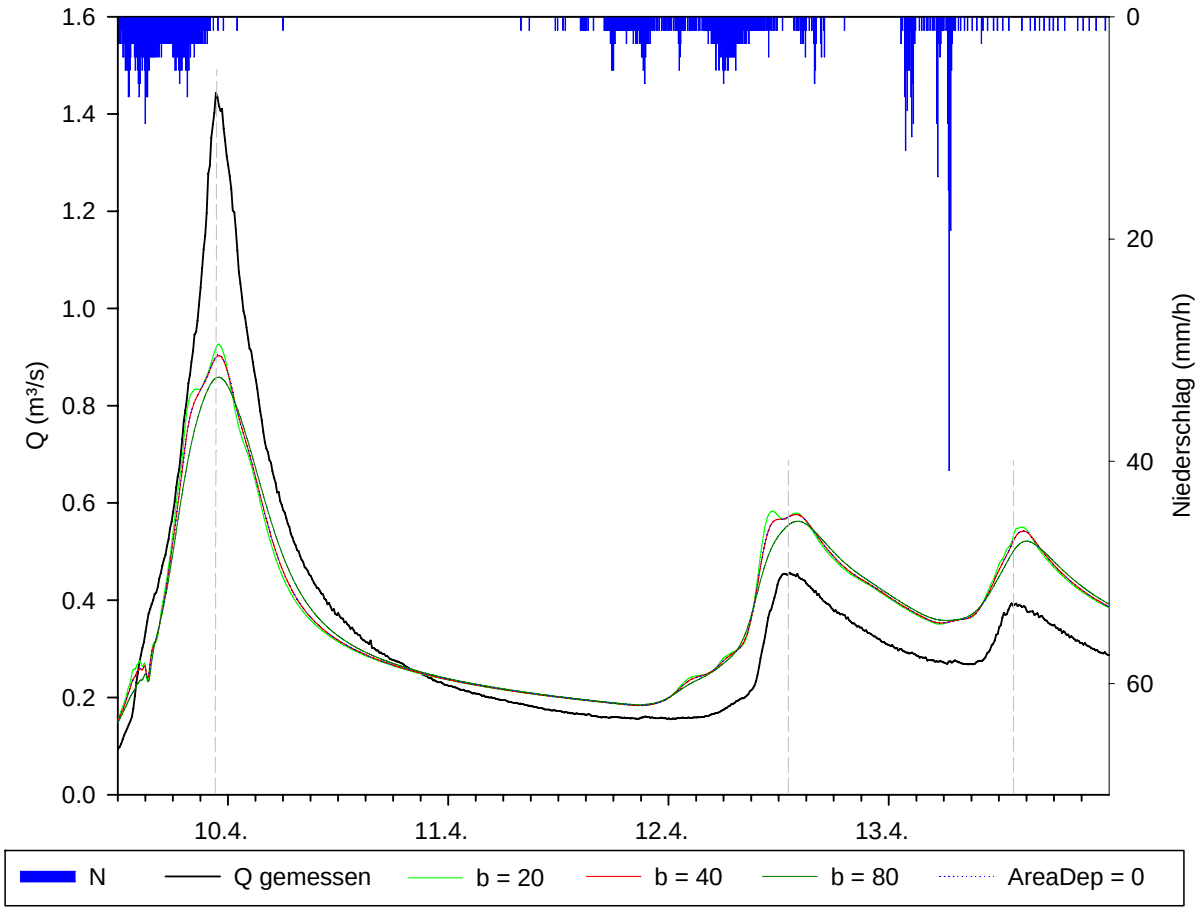


Abb. 5.7 Einfluss des Abfluss-Konzentrations-Parameters b

Insgesamt zeigt sich, dass der Einfluss von Veränderungen der Einheitsganglinien der Teileinzugsgebiete bei der Modellierung des Dragonja-Gebietes einen untergeordneten Einfluss auf die Modellergebnisse hatte, obwohl die Zahl der Teileinzugsgebiete mit 34 (ohne das Rokava-Gebiet) deutlich niedriger lag als in den meisten bisherigen Anwendungen. Bei einer weiteren Reduzierung der Zahl der Teileinzugsgebiete wäre mit einer zunehmenden Bedeutung der Einheitsganglinie zu rechnen. Bei einer größeren Zahl von Teileinzugsgebieten hingegen wird der Abstand zwischen dem Ort der Abflussbildung und dem Gerinne sowie das Verhältnis zwischen Konzentrationszeit im Teileinzugsgebiet und der Fließzeit im Gerinne kleiner, sodass die Bedeutung der Abflusskonzentration schwindet. So haben LANGE ET AL. (2001) bei der Modellierung eines 250km² großen Einzugsgebiet in den Judean Mountains in Israel mit 488 Teileinzugsgebieten wegen der geringen Größe der Teileinzugsgebiete den gebildeten Abfluss direkt dem Gerinne zugefügt.

5.3.3 Übergang vom ZIN- zum Routing-Zeitschritt

Der Zeitschritt für das Channel-Routing kann sich von dem des restlichen ZIN-Modells unterscheiden, wobei der ZIN-Zeitschritt ein Vielfaches des Routing-Zeitschritts sein muss. Den im Abflusskonzentrations-Teil des Modells für das Routing erzeugten Input unverändert zu verwenden stellt eine potenzielle Quelle numerischer Instabilität dar. Es muss also ein Übergang zwischen den unterschiedlichen Zeitschritten geschaffen werden. Bisher geschah dies nach der Abflusskonzentration, indem der Abfluss eines ZIN-Zeitschritts gleichmäßig auf die entsprechenden Routing-Zeitschritte verteilt wurde. Da mit der EV I - Verteilung die Möglichkeit besteht, für jeden Zeitpunkt einen Wert zu erhalten, wurde der Zeitschritt-Übergang in die Abflusskonzentration verlegt.

5.4 Channel-Routing

Das Channel-Routing im ZIN-Modell enthält neben dem eigentlichen Muskingum-Cunge-Verfahren auch die Repräsentation von Transmission-Losses. Deren Konzeption und Implementation werden ausführlich von LEISTERT (2005) beschrieben und ist nicht Bestandteil dieser Arbeit.

5.4.1 Implementation des Muskingum-Cunge-Verfahrens

Das Abbruchkriterium der in Abschnitt 4.1.5 beschriebenen und in Abbildung 5.8 dargestellten Optimierungsschleife war bisher über die Differenz von Q und Q_{ref} formuliert, der entsprechende Schwellenwert für einen Differenz-Abfluss in m^3/s wurde direkt im Quellcode eingegeben. Der Nachteil dieser Vorgehensweise ist, dass ein fester Schwellenwert für jedes Segment zu jedem Zeitschritt gilt. Ein sehr großer Wert, der dabei für den Gebietsauslass sinnvoll ist, wäre als Toleranz für ein Segment des Oberlaufs wahrscheinlich inakzeptabel groß. Dies führt dazu, dass die Schleife in den meisten Fällen nur einmal durchlaufen wird, was dem linearen Muskingum-Cunge-Verfahren entspricht.

Wiederum könnte ein kleiner Schwellenwert, der für den Oberlauf sinnvoll wäre, bei größeren Abflüssen dazu führen, dass die geforderte Genauigkeit nie erreicht wird und das Programm in einer Endlosschleife gefangen bleibt. Das Gleiche gilt sinngemäß für die zeitlichen Variationen des Abflusses.

Aus diesen Gründen wurde die Abbruchbedingung über die relative Abweichung zwischen Q und Q_{ref} formuliert. Dafür wird ein zulässiger Fehler E_{curr} festgelegt. Solange die Ungleichung

$$\left|Q_{ref} - Q\right| > \left|Q \cdot E_{curr}\right| \qquad (5.12)$$

erfüllt ist, wird Q_{ref} der Wert von Q zugewiesen und die Schleife ein weiteres mal durchlaufen. In numerisch ungünstigen Situationen kann es dazu kommen, dass die Schleife bei dem verwendeten zulässigen Fehler nicht terminiert. Für diesen Fall wird die Zahl der bisher durchlaufenen Iterationen mitgezählt. Überschreitet diese die Schwelle von 500, wird angenommen, dass das System nicht zu einer Lösung konvergiert. Der zulässige Fehler wird dann um 10 % erhöht und es wird erneut versucht, eine Lösung zu finden.

Jeweils vor Beginn der Schleife wird der Fehler wieder auf einen Ursprungswert zurückgesetzt. Für die in dieser Arbeit durchgeführten Modellläufe wurde für diesen Fehler ein Wert von 0,1 % verwendet, mit dem eine geringe Abweichung bei einer akzeptablen Zahl von Iterationen erreicht wurde. Die zulässige Zahl von Iterationen wurde dabei in keinem Fall erreicht.

Um das Muskingum-Cunge-Verfahren nicht nur auf ein einzelnes Gerinnesegment, sondern ein Gewässernetz anwenden zu können, müssen die einzelnen Gerinneabschnitte sowie laterale Zuflüsse in eine räumliche Beziehung zueinander gesetzt werden. Die Verarbeitung jedes Segments kann erst erfolgen, wenn alle oberhalb gelegenen Abschnitte für den betreffenden Zeitschritt bereits verarbeitet sind. Dies wird im TRAIN-ZIN-Modell über eine der Verarbei-

tungsreihenfolge entsprechenden Nummerierung der Segmente erreicht, die der Anwender nach dieser Vorgabe erstellen muss.

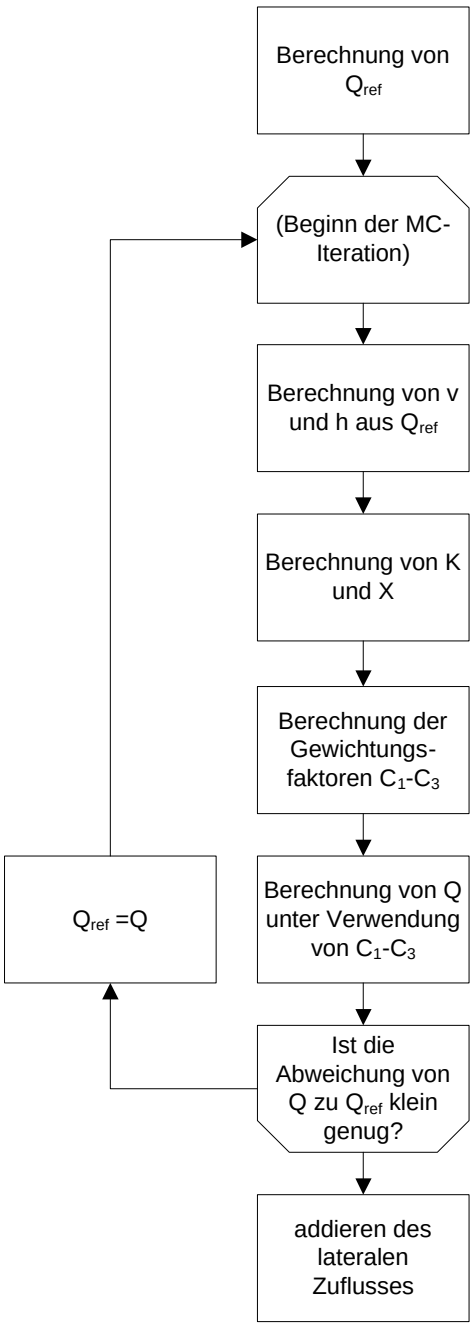


Abb. 5.8 Iterative Berechnung des Abflusses nach dem nicht-linearen Muskingum-Cunge-Verfahren

Die Zuordnung erfolgt in Tabellenform mit einer Zeile für jedes Segment. In den Spalten steht jeweils der Index

A. des Segments selbst

B. des obenliegenden Segments

C. des nachfolgenden Segments

D. des ersten Zuflusses

E. des zweiten Zuflusses

sowie

F. das Gefälle

G. die Länge

H. die Breite

I. der Typ

J. eine 1, wenn das Segment in der Ausgabe enthalten sein soll, sonst eine 0.

Die Nummerierung bezieht sich auf die Segmente selbst, nicht etwa auf die dazwischenliegenden Knotenpunkte. Fehler bei der Aufstellung der Zuordnung werden vom Modell nicht erkannt und müssen vor der Anwendung ausgeschlossen werden.

Die Verwendung der Spalte J zur Ausgabe wurde neu implementiert. Bisher war eine Ausgabe nur für ein einziges Segment möglich. Wurde der Abfluss an mehreren Stellen benötigt, musste dafür für jedes Segment ein extra Modelllauf durchgeführt werden.

Eine weitere Neuerung ist, dass die Gerinneeigenschaften der verschiedenen Gerinnetypen (Spalte I) nicht mehr im Quellcode stehen, sondern aus einer Datei eingelesen werden. Damit ist es für einen Anwender leicht möglich, eigene Gerinnetypen zu definieren. Die Zahl der vorhandenen Typen wird beim Einlesen automatisch erkannt. Zu beachten ist, dass der Typ 1 für Rohre reserviert ist, alle weiteren Typen werden als offene Gerinne interpretiert.

Für die Berechnung des Abflusses nach der Gleichung 4.9 und des Referenzabflusses nach der Gleichung 4.15 muss festgelegt werden, an welcher Stelle die Zuflüsse aus obenliegenden Gerinnen und wo die lateralen Zuflüsse hinzuaddiert werden. Bisher geschah dies innerhalb der Schleife aus Abbildung 5.8 als zusätzliche Terme der Gleichung 4.9. Für die Zuflüsse I_x wurde der Zufluss nur aus dem obenliegenden Segment verwendet, für die Abflüsse Q_x der Fluss aus dem Segment heraus ohne die diesem Segment zugeordneten lateralen Zuflüsse, aber mit den

dem unteren Segmentende zugeordneten Seitenarmen. Die Berechnung von Q_{t+1} erfolgte dann nach

$$Q_{t+1} = C_1 I_{t+1} + C_2 I_t + C_3 (Q_t - Q_{lat_t}) + Q_{lat_{t+1}} + Q_{trib1} + Q_{trib2} \qquad (5.13)$$

Ist der für das Abbruchkriterium gewählte akzeptable Fehler so groß, dass die Schleife nur ein einziges mal durchlaufen wird, entspricht dies dem linearen Verfahren. Bei jedem weiteren Schleifendurchlauf würden die Gewichtungsfaktoren C_1 bis C_3 allerdings aus dem *gesamten* Abfluss einschließlich aller Zuflüsse berechnet, aber nur auf einen Teil des Abflusses angewendet.

FREAD (1992) beschreibt die Verwendung eines weiteren Gewichtungsfaktors C_4 zur Berücksichtigung des lateralen Zuflusses. Die laterale Zuflusskomponente geht dann als zusätzlicher Additionsterm in Gleichung 4.9 ein und wird berechnet als

$$Q_{lat} \cdot C_4 = Q_{lat} \cdot \frac{\Delta t \cdot \Delta x}{2K(1-X) + \Delta t} \qquad (5.14)$$

Für die vorliegende Arbeit wurde allerdings ein vereinfachter Weg gewählt, indem die lateralen Zuflüsse von der Routing-Schleife ausgenommen blieben und erst nach deren Ende dem berechneten Abfluss hinzugerechnet wurden. Der eigentlich über die Segmentlänge verteilte, linienhafte laterale Zufluss wird dem Gerinne damit punktförmig erst am unteren Ende zugeführt. Dies korrespondiert mit dem für die Teileinzugsgebiete verwendeten Unit-Hydrograph-Ansatz, der ebenfalls den Abfluss eines Gebietes an einem Punkt beschreibt.

Die Zuflüsse aus anderen Segmenten wurden jeweils als Teil des Zuflusses I_t bzw. I_{t+1} betrachtet, also dem oberen Ende des zugehörigen Segments zugeordnet.

Die einzigen Bedingungen für die Nummerierung der Segmente sind, dass jeder Vorgänger jeden Segmentes, also Zufluss 1, Zufluss 2 und das im Hauptgerinne vorhergehende Segment, einen niedrigeren Index als das aktuelle Segment hat und dass die Nummerierung bei eins beginnend lückenlos erfolgt. Die Beachtung der Einteilung in Hauptgerinne und Zuflüsse dient dabei der Orientierung, macht für die Berechnung aber keinen Unterschied.

5.4.2 Längen der Gerinnesegmente

Um dem Anwender des TRAIN-ZIN-Modells die Entscheidung für die Wahl der Segmentlänge zu erleichtern, wurde eine Ausgabe der nach Gleichung 4.16 geeigneten Länge für jedes

Gerinne-Segment für die durchschnittliche Fließgeschwindigkeit sowie für die maximale Fließgeschwindigkeit implementiert.

5.5 Warmstart

Im Laufe der Modellierungsarbeiten zeigte sich, dass die Vorfeuchte im Modell einen wesentlichen Einfluss auf die Modellergebnisse der ersten Tage bis Wochen hat. Um die Dauer der Warmlaufphase zu verkürzen, wurde die Möglichkeit geschaffen, ein Grid mit relativen Bodenfeuchten einzulesen und so zellenweise Startwerte zu setzen. Dafür bietet es sich an, die Ausgabe aus einem früheren Modelllauf zu verwenden. Eine einfache Möglichkeit, ein passendes Start-Grid auszuwählen, liegt im Vergleich der Basisabflüsse von Modell und Messung. Ob damit die richtige räumliche Verteilung der Feuchte wiedergegeben wird, ist allerdings ungewiss.

Für das TRAIN-Modell wurde ebenfalls ein Start mit definierten Werten bereits für den ersten Tag implementiert, indem das entsprechende Grid schon vor dem ersten Aufruf des TRAIN-Modells mit den der ZIN-Bodenfeuchte entsprechenden Werten gefüllt wird.

5.6 Wasserbilanz

Die Berechnung der Wasserbilanz eines Modells ist ein wichtiges Werkzeug zur Kontrolle, ob das Modell fehlerfrei arbeitet. Durch die Einbeziehung der noch fehlenden Komponenten in die Wasserbilanz, insbesondere der Speicheränderung, kann diese Kontrolle nun durchgeführt werden. Die Ausgabe von Werten für Niederschlag, ETA, Tiefeninfiltration, Speicheränderung (Boden und Anfangsverlustspeicher), Abfluss sowie der errechnete Wasserbilanzfehler erfolgt in eine Datei in Tagesschritten, wobei alle Werte in mm angegeben werden, um einen schnellen Überblick über die Größenordnung der einzelnen Komponenten zu bekommen.

5.7 Weitere programmtechnische Veränderungen

5.7.1 Einlesen von Grids

Das Einlesen der Grids erfolgte bisher zuerst in einen `vector` und von diesem in die programminterne Matrix. Da das Einlesen aus Dateien von links nach rechts und von oben nach unten erfolgt und die Zählung der y-Koordinaten von Null aufsteigend erfolgte, wurde innerhalb

des Programms ein Koordinatensystem mit dem Ursprung in der linken oberen Ecke verwendet, während bei den gewöhnlich mit einem GIS erzeugten Grid-Dateien der Ursprung in der linken unteren Ecke liegt. Da alle Grids auf die gleiche Weise eingelesen und verarbeitet wurden, ergab sich daraus keine fehlerhafte Verarbeitung. Für die neu eingeführte Verarbeitung von Niederschlagsstationen ergab sich aber die Notwendigkeit, die Positionen der Stationen in Grid-Koordinaten anzugeben. Dazu wäre ein Benutzer gezwungen gewesen, die richtige Koordinatentransformation in der Form

$$y' = ySize - y \qquad (5.15)$$

durchzuführen, was eine potenzielle Fehlerquelle darstellt. Ein weiterer Grund, das Einlesen umzustrukturieren, war eine angestrebte Straffung des Quellcodes in Verbindung mit schnellerer Laufzeit. Das Einlesen erfolgte bisher in mehreren Schritten wie in Abbildung 5.9 dargestellt.

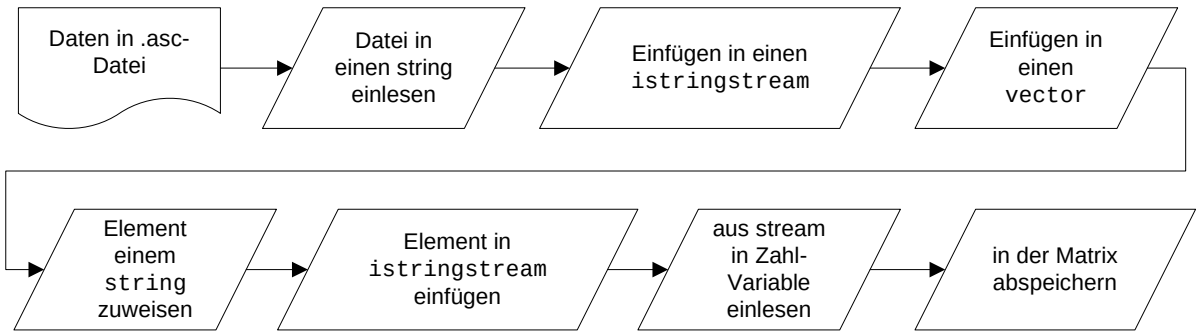


Abb. 5.9 bisheriges Verfahren zum Einlesen eines Grids aus einer Datei

Da es ohne Weiteres möglich ist, direkt aus dem mit der Eingabedatei verknüpften Inputstream zu lesen, wurde das Vorgehen zu dem in Abbildung 5.10 geändert.

Die einzige Einschränkung, die dieses Vorgehen mit sich bringt, ist, dass es nicht mehr möglich ist, zum Einfügen in die Matrix auf beliebige Elemente der Eingabe in Form eines `vectors` zuzugreifen. Um also im Koordinaten-System der Eingabedatei zu bleiben, müssen alle *y* in *absteigender* Reihenfolge durchlaufen werden, sowie für jedes *y* alle *x* in *aufsteigender* Reihenfolge.

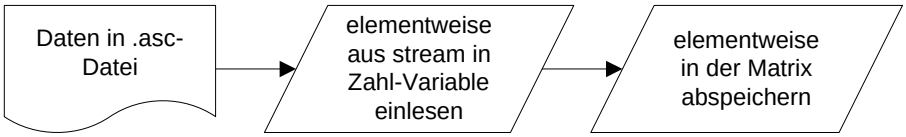


Abb. 5.10 neues Verfahren zum Einlesen eines Grids aus einer Datei

5.7.2 Ausgabe von Grids in Dateien

Die komplette Implementierung der Ausgabe von Grids erfolgte bisher jeweils an der Stelle, an der das Grid verarbeitet wurde. Dadurch war der entsprechende Programmcode mehrfach in sehr ähnlicher oder gleicher Form im Programm vorhanden. Da dies die Wartung und Übersichtlichkeit erschwert wie in Abschnitt 3.1.4 beschrieben, wurde die Methode `Grid::writeGrid(...)` ausgebaut und alle Ausgaben von Grids auf diese eine Methode „umgeleitet".

Eine wesentliche Verbesserung stellt die Ausgabe im ASCII-Grid-Format dar, die sich ohne weitere Bearbeitung in ArcGis öffnen lassen. Dazu sind zu jedem Grid vollständige header-Informationen nötig. Diese werden bei der Erstellung jeden Grids aus einer Datei eingelesen, die in der Steuerungsdatei unter dem Schlüsselwort „header" angeben ist. Dies kann entweder eine gesonderte Datei sein, die lediglich die sechs header-Zeilen enthält oder jedes ohnehin verwendete Grid mit dem richtigen header. Die sechs header-Elemente werden in Instanzvariablen in jedem Grid gespeichert und bei der Ausgabe an den Anfang der Datei geschrieben. Der als Argument zu übergebende Dateiname (einschließlich Pfad) wird mit der Dateinamenerweiterung .asc versehen.

Hierbei wird die Gleichheit aller Grids bezüglich des headers vorausgesetzt. Wird eine falsche header-Information verwendet, kann das Grid entweder gar nicht in ArcGis dargestellt werden oder erscheint verzerrt und/oder verschoben.

5.7.3 `char`-Arrays und `strings`

Alle im Programm vorkommenden `char`-Arrays (C-strings) wurden auf C++-strings umgestellt. Die C-strings wurden größtenteils dazu verwendet, Dateinamen bzw. Pfadangaben zu behandeln. Ein Vorteil von C++-strings ist, dass sie in ihrer Länge flexibel sind. So kann es

nicht vorkommen, dass über das Ende eines Arrays hinaus geschrieben wird mit den in Kapitel 3.1.1 beschriebenen Folgen. Auch die Handhabung z. B. beim Kopieren und Vergleichen ist wesentlich einfacher. Die Entsorgung nicht mehr benötigter `strings` wird vom System übernommen. Da die `char`-Arrays nicht an geschwindigkeitskritischen Stellen vorkamen, sind keinerlei Nachteile zu erwarten.

5.7.4 Unreferenzierter Programmcode

Unreferenzierter Programmcode, also Teile, die von keiner Stelle aus aufgerufen werden, erschwert die Übersicht im Quellcode erheblich. So existierten häufig überladene Methoden, von denen aber eine oder sogar mehrere nicht referenziert waren oder Methoden, die Überreste aus früheren Anwendungen waren und keinem erkennbaren Zweck mehr dienten. Während unreferenzierte Variablen vom Compiler erkannt werden, ist dies bei Methoden nicht der Fall. Um unreferenzierte Methoden zu identifizieren, wurden diejenigen, bei denen eine Verwendung nicht unmittelbar offensichtlich war, auskommentiert. Ergab ein Versuch, das Projekt zu kompilieren, dass die Methode tatsächlich nicht aufgerufen wird, wurde sie auskommentiert und an das Ende der Klassen-Datei verschoben. Da nicht immer geprüft werden konnte, ob vielleicht wertvoller Code enthalten war, wurde auf das Löschen verzichtet.

Auf diese Weise konnte die Menge an Programmtext, die der Programmierer überblicken muss, erheblich reduziert werden. So umfasste die Klasse `Runoff` ursprünglich etwa 800 Zeilen. Von diesen wurden etwa 600 auskommentiert und ans Dateiende verschoben, während ca. 200 Zeilen übernommen und ca. 100 neue hinzugefügt wurden, um die Funktionalität zu erweitern. Unabhängig von inhaltlichen Bearbeitungen ist hier schon die Reduktion der Menge an Programmtext ein Gewinn, da es die Übersichtlichkeit erheblich verbessert und so dazu beiträgt, den Programmtext wartbar und verstehbar zu halten.

Unreferenzierte Variablen und Parameter werden vom Compiler identifiziert und wurden zum größten Teil ebenfalls entfernt, um die Übersichtlichkeit zu erhöhen.

5.7.5 Lesen und Schreiben von Dateien

Das Öffnen einer Datei zum Lesen oder Schreiben ist zuerst als Versuch zu verstehen, ob Lesen oder Schreiben von dieser Datei möglich ist. Eine Überprüfung, ob das Öffnen erfolgreich war oder ob während des Lesens oder Schreibens ein Fehler auftrat, fand in der ursprünglichen Version nicht statt.

Fehler beim Lesen können dadurch entstehen, dass die zu öffnende Datei nicht gefunden wird, dass sie in einer anderen Anwendung geöffnet und gesperrt ist, dass der Benutzer über keine Leserechte für die Datei verfügt, dass versucht wird, ein nicht kompatibles Element in eine Variable einzulesen oder dass versucht wird, über das Ende der Datei hinaus zu lesen.

Nach einem Lesefehler wird der Input-Stream als fehlerhaft markiert und ein weiteres Lesen ist nicht mehr möglich, bis die Fehler-Markierung ausdrücklich entfernt wird. Wird ein Lesefehler ignoriert, enthalten die Variablen, in die gelesen werden sollte, anschließend unvorhersehbare Werte. Es ist nicht sichergestellt, dass der Fehler bemerkt wird – in ungünstigen Fällen könnte der Modelllauf ohne offensichtlichen Fehler beendet werden und die Ausgabe scheinbar richtige Werte enthalten. Eine Überprüfung, ob der Input-Stream erwartungsgemäß funktioniert, ist also unerlässlich und wurde an allen Stellen, an denen eine Datei geöffnet wird, eingefügt. Außerdem wurde eine Überprüfung an vielen Stellen eingefügt, an denen ein Fehler während des Lesens (wie das Erreichen des Endes der Datei) möglich erschien.

Beim Schreiben in Dateien können Fehler dadurch entstehen, dass die zu öffnende Datei durch eine andere Anwendung geöffnet und geschützt ist, dass der Benutzer über keine Schreibrechte für die Datei verfügt oder dass der Pfad (Ordner), in den geschrieben werden soll, nicht vorhanden ist. Fehler bei der Datenausgabe mussten bislang vom Benutzer anhand von Änderungsdatum und -zeit der Ausgabedatei erkannt werden, um nicht die Ergebnisse eines früheren Modelllaufs mit aktuellen Ergebnissen zu verwechseln. War beispielsweise die Ausgabe des letzten Laufs noch in einem Programm (z. B. MS Excel) geöffnet, um die Ergebnisse zu analysieren, während das Modell versuchte, eine neue Ausgabedatei mit gleichem Namen zu schreiben, musste der gesamte Modelllauf wiederholt werden, um die durch erfolgloses Schreiben verlorenen Daten wiederzuerhalten.

Um dem Benutzer die Möglichkeit zu geben, die Ursache für den Schreibfehler zu beheben, ohne den u. U. sehr zeitaufwendigen Modelllauf zu wiederholen, wurde eine Methode „`checkWrite`" in der `Controller`-Klasse implementiert. Diese erhält als Argument den zu prüfenden Dateinamen mit vollständigem Pfad. Solange es nicht möglich ist, in die betreffende Datei zu schreiben, wird eine Meldung auf dem Bildschirm ausgegeben und der Benutzer hat die Wahl, die Ursache des Problems zu beheben (also z. B. die Datei in der anderen Anwendung zu schließen), oder das Programm zu beenden. Nur bei erfolgreichem Schreibversuch wird also zur aufrufenden Stelle zurückgekehrt, sodass der Output-Stream hier auf jeden Fall geöffnet werden kann.

5.7.6 Schnittstelle zwischen Abflusskonzentration und Routing

Bedingt durch die Entwicklung des ZIN-Modells aus unabhängig voneinander lauffähigen Modulen war die Schnittstelle zwischen Abflusskonzentration und Routing bisher über das Schreiben/Lesen einer Datei verwirklicht. Da in der jetzigen Version alle Module integriert sind, bot es sich an, zur Verkürzung der Laufzeit eine Übergabe innerhalb des Modells zu realisieren. Um dabei ein aufwendiges Kopieren des Vektors `latq`, der die Daten aus der Abflusskonzentration enthält, zu vermeiden, wurde die Methode getLatqRef() mit dem Rückgabetyp einer Referenz auf den Vektor (`vector<vector<double>> &`) implementiert. Diese Referenz wird von der Methode `Execution::run()` abgerufen und ans Routing weitergegeben.

5.7.7 Einlesen der Gerinneparameter

Die Gerinneparameter wurden bislang im Quellcode des Routing angegeben. Dies erschwerte das Ändern der Parameter und insbesondere das Hinzufügen neuer Gerinnetypen. Es wurde stattdessen das Einlesen der Parameter aus einer Datei implementiert, deren Name in der Steuerungsdatei unter dem Schlüsselwort „typeProps" anzugeben ist. Die Zahl der Gerinnetypen wird dabei automatisch erkannt. Der Gerinnetyp 1 wird als Rohr interpretiert (siehe GAßMANN 2007), alle Weiteren als offene Gerinne.

5.7.8 Datumsformat

Zur Erhöhung des Benutzer-Komforts wurde an mehreren Stellen die Verwendung von Tageszahlen (x-ter Tag des Jahres) durch ein allgemein verständliches Datumsformat im Format dd.mm.yyyy ersetzt. Dies erfolgte für Start- und Enddatum der Modellierung in der Steuerungsdatei, für die Niederschlagsinputdaten und für die Ausgabe des Abflusses.

Dazu wurden einige Methoden in der `Controller-`Klasse angelegt, die allerdings keinen logischen Zusammenhang zum Rest dieser Klasse aufweisen.

5.7.9 Optionale Verwendung von TRAIN

Da sich die Verwendung des TRAIN-Modells je nach Setup und Computer-System als sehr zeitintensiv herausstellte, wurde die Möglichkeit geschaffen, die Ausführung des TRAIN-Teils zu überspringen und stattdessen die Werte vorheriger Simulationen zu verwenden. Die Ent-

scheidung, ob TRAIN für den betreffenden Tag ausgeführt wird, erfolgt dabei nach dem in Abbildung 5.11 dargestellten Verfahren.

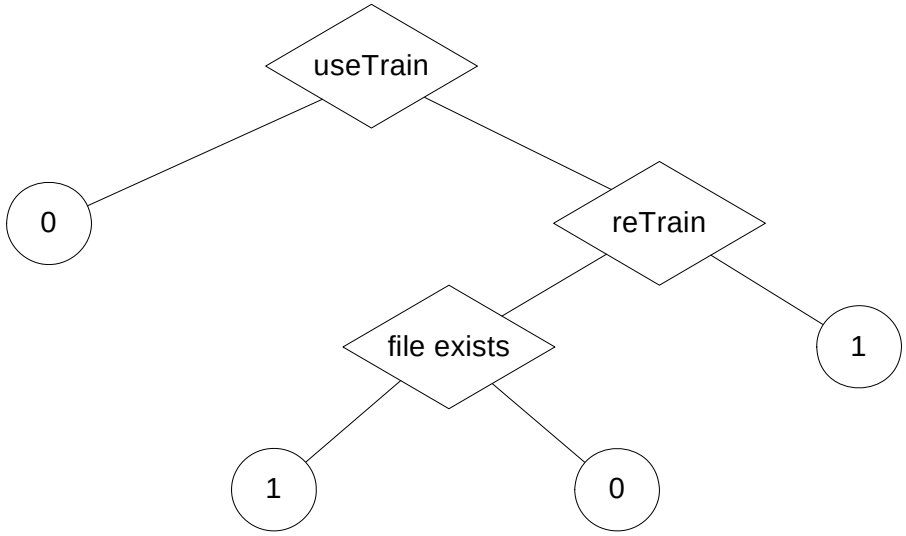


Abb. 5.11 Entscheidungsbaum für die Verwendung des TRAIN-Modells. (Bei positiver Entscheidung nach rechts, sonst nach links. *useTrain*: Verwendung von TRAIN ein/aus. *reTrain*: TRAIN jedenfalls neu durchlaufen. *file exists*: Datei mit Verdunstungswerten ist vorhanden. 0: Train wird nicht gestartet. 1: TRAIN wird gestartet)

5.8 Fazit

Der Funktionsumfang des TRAIN-ZIN-Modells konnte erheblich erweitert werden. Niederschlagsdaten können dem Modell nun auch als Stationsdaten übergeben werden. Die Verteilung der punktförmigen Messungen auf das Gebiet kann wahlweise mit den einfacheren Thiessen-Polygonen oder dem Inverse-Distance-Weighting Verfahren erfolgen. Bei diesem ist eine Höhenkorrektur der Daten möglich.

Der synthetische Unit-Hydrograph-Ansatz für die Abflusskonzentration wurde erweitert und auf die Extremwertverteilung I umgestellt. Mittlere Hangneigung der Teil-Einzugsgebiete und die Teileinzugsgebietsflächen werden verwendet, um eine Standardkurve an die einzelnen Gebiete anzupassen. Die Implementation wurde dabei so verändert, dass ein Überlaufen der Speicherstrukturen zuverlässig ausgeschlossen wird.

Im Channel-Routing wurde die Muskingum-Cunge Iteration überarbeitet und durch die Einführung eines relative Fehlers als Abbruchkriteriums und eine Neuordnung der Zuflüsse die Einhaltung des nicht-linearen MC-Verfahrens sichergestellt.

Durch die Möglichkeit, ein Bodenfeuchte-Grid als Startwert zu verwenden, kann die Warmlaufphase bei uneinheitlicher Vorfeuchte deutlich verkürzt werden.

Durch die Erstellung einer Benutzer-Schnittstelle konnte die Bedienung des Modells wesentlich vereinfacht werden. Auch Benutzer ohne Programmierkenntnisse können das Modell nun anwenden.

Aus der programmtechnischen Perspektive konnte außerdem die Zuverlässigkeit sowie die Geschwindigkeit des Modells erheblich verbessert werden.

6 Das Dragonja-Gebiet

6.1 Lage und Klima

Das Dragonja-Einzugsgebiet liegt auf der Halbinsel Istrien an der Grenze zwischen Slowenien und Kroatien (Abbildung 6.1). Es umfasst eine Fläche von 92 km² und Höhenstufen von 490 m bis zum Meeresniveau an der Mündung in den Golf von Venedig. Das Untersuchungsgebiet umfasst nur den oberen Teil mit 51 km² ab der Einmündung des Seitenflusses Rokava. Da Abflussdaten nur für den oberen Dragonja-Pegel vorlagen, beziehen sich die Angaben bei Abflüssen nur auf diesen Teil des Gebietes.

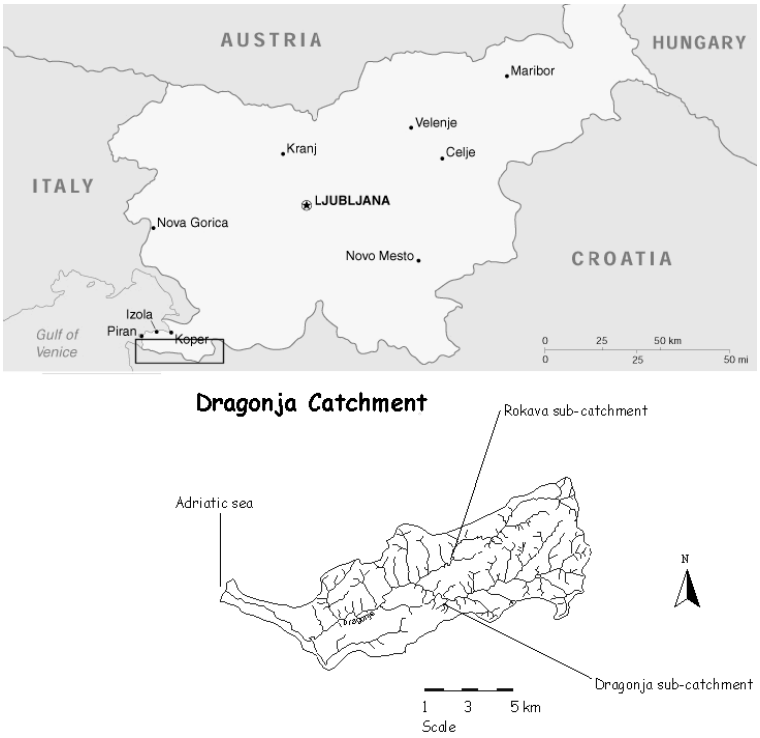


Abb. 6.1 Lage des Dragonja-Gebiets (KEESTRA 2005)

Angaben zu Temperaturen (Lufttemperaturen auf Meereshöhe reduziert) und Niederschlagsmengen nach MARDEŠIĆ ET AL. (1962) für die Periode 1925 - 1940 sind in Tabelle 6.1 dargestellt. Dabei nehmen die Temperaturen sowohl im Sommer als auch im Winter zum Landesinneren hin ab, während der Niederschlag zunimmt.

Tabelle 6.1 Klimatische Bedingungen im Dragonja-Gebiet (nach Mardešić et al., 1962)

	Lufttemperatur (˚C)	Niederschlag (mm)	Meerestemperatur (˚C)
Winter	-	150 - 300	10
Januar	2 - 4	-	-
Sommer	-	-	25
Juli	22 - 24	-	-
Jahresmittel	12 - 14	800 - 1200	-

Der Jahresgang des Niederschlags ist dabei nicht stark ausgeprägt mit einem schwachen Maximum im Herbst (VAN DER TOOL 2007). In der Klimaklassifikation von Köppen gehört das Gebiet demnach zu den feuchten, warm gemäßigten Klimaten mit heißen Sommern (Cfa).

In der Klimaklassifikation nach THORNTHWAITE (1948) gehören die höheren Lagen des Gebiets zum Typen $B_3^4 B_2^{'} r b_3^{'}$, der untere Gebietsteil in Meeresnähe zum Typ $B_1^2 B_2^{'} s b_3^{'}$ (BREU, 1970 - 1989). Die Typen B_1 bis B_4 (erster Buchstabe) sind dabei in zwei Klassen zusammengefasst und bezeichnen die humiden Klimate. Der folgende Buchstabe *r* steht für ein geringes oder kein jahreszeitliches Wasserdefizit, *s* für ein mäßiges sommerliches Wasserdefizit. b_2' bezeichnet ein wenig maritimes, b_3' ein mäßig maritimes Klima.

6.2 Geologie, Topografie und Böden

Der größte Teil des Gebietes liegt auf eozänem Flysch, einem aus Lagen weichen, kalkhaltigen Tonsteins und Sandstein aufgebauten marinen Sediment. Im unteren Teil des Einzugsgebietes besteht der Südhang des Dragonja-Flusses aus Kalkstein aus der Oberkreide. Die Bergrücken bilden flache Plateaus, die Täler sind eng mit steilen Hängen.

Der Boden besteht zum größten Teil aus Rendzina. Auf den Plateaus ist diese teilweise über einen Meter mächtig, auf den Hängen teilweise nur einige Dezimeter oder gar nicht vorhanden. Die mehrere Meter mächtigen Auenböden im Tal liegen auf karbonatischem Alluvium. (GLOBEVNIK 1998)

7 Eingangsdaten und Modellparameter

7.1 Vorhandene Daten und Datenaufbereitung

Alle Daten wurden von Christiaan van der Tool von der Freien Universität Amsterdam zur Verfügung gestellt.

Im oberen Dragonja-Gebiet gibt es zwei Pegel. Der Pegel der Dragonja liegt ca. 500 m, der der Rokava etwa 300 m flussaufwärts des Zusammenflusses der beiden Flüsse (siehe Abbildung 7.2). Abflussdaten lagen nur für den Pegel Dragonja vor, sodass sich alle Angaben hinsichtlich des Abflusses nur auf dieses Teileinzugsgebiet beziehen.

Die Zeiträume, für die alle für die Modellierung notwendigen Daten gleichzeitig vorlagen, lagen im Zeitraum zwischen Februar 2002 und Dezember 2003 (Abbildung 7.1).

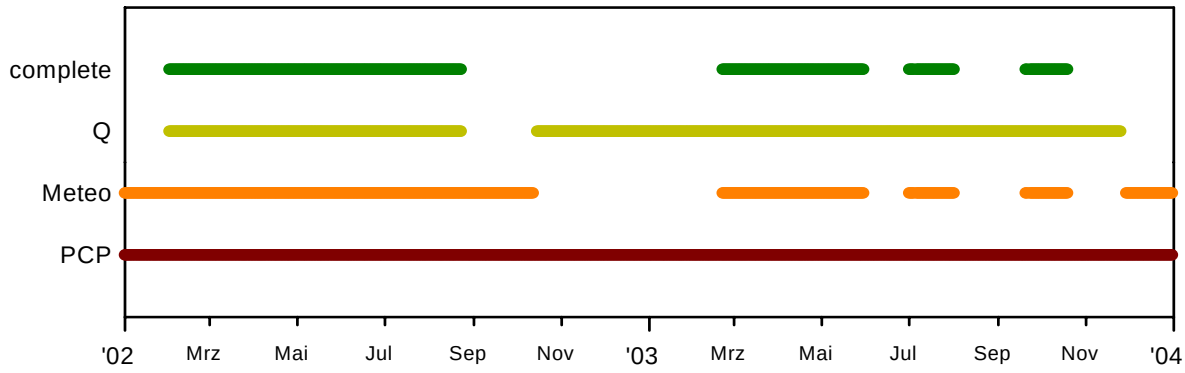


Abb. 7.1 Verfügbarkeit von Daten (von unten: Niederschlag, weitere meteorologische Daten, Abfluss, alle Daten gleichzeitig)

7.1.1 Pegel Dragonja

Für den Pegel Dragonja lagen Daten für die Zeit zwischen dem 1.10.2000 und dem 18.4.2004 in Zehn-Minuten-Intervallen vor. Die Pegelstände waren dabei als Ganzzahl-Werte in 10^{-1} mm gespeichert, so dass ein durch die Speicherung verursachter Verlust an Genauigkeit ausgeschlossen werden kann. Für die Angabe des Zeitpunktes der jeweiligen Messung war das nicht der Fall. Die Zeiten lagen im Format des Programms „Matlab" vor. Dieses gibt die Zahl der Tage nach einem willkürlich festgelegten Zeitpunkt an. Da die Matlab-Software nicht zur Verfügung stand, wurde mit Microsoft Excel gearbeitet, das das gleiche Verfahren zur Repräsentation von Zeiten verwendet. Allerdings liegt der Bezugszeitpunkt in Excel beim 1.1.1900

(„Tag 1") der von Matlab um 693960 Tage früher, was rechnerisch dem 1. Januar des Jahres 0 entspricht.

Da dieser Bezugswert sechs Stellen hat und die Zeitwerte mit einer Genauigkeit von 8 signifikanten Stellen gespeichert wurden, blieben zur Darstellung der Uhrzeit eines Tages zwei Dezimalstellen. Der kleinste mögliche Schritt von 0,01 entspricht dabei einer Zeit von 14:24 min., sodass eine eindeutige Zuordnung zu Zeitpunkten im Zehn-Minuten-Abstand nicht ohne weiteres möglich war.

Für diese Zuordnung wurde ein Programm erstellt, das jeden Messwert einem Zehn-Minuten-Schritt zuordnet. Da in den Zeitwerten regelmäßig ganze Zahlen auftraten, wurde davon ausgegangen, dass diese eindeutig der Uhrzeit 0:00 zugeordnet werden können, da hier keine Nachkommastellen bei der Darstellung auftreten. Des weiteren reichen zwei Dezimalstellen zur exakten Darstellung der Zeiten 6:00 (0,25), 12:00 (0,5) und 18:00 (0,75). Davon ausgehend wurden Abschnitte zwischen zwei als bekannt angenommenen Zeiten betrachtet. Stimmte die Zahl der vorhandenen Werte multipliziert mit 10 Minuten mit der Zeitdifferenz überein, wurden die Messwerte gleichmäßig in 10 Minuten Schritten auf diesen Zeitraum verteilt. Andernfalls ergab sich die Zahl fehlender Messwerte. Für diese wurden zusätzliche Zeitwerte iterativ nach derjenigen Stelle eingefügt, an der die Differenz zwischen künstlich erzeugten 10-Minuten-Schritten und den ursprünglich vorhandenen Zeitwerten das letzte mal kleiner als 10 Minuten war. Auf diese Weise konnte erreicht werden, dass das arithmetische Mittel der Abweichung zwischen ursprünglichem und ermitteltem Zeitwert bei 2:55 min lag. Dieser Fehler liegt unterhalb der verwendeten Auflösung für die Ausgabe des Modells von 5 Minuten.

Für die Wasserstands-Abfluss-Beziehung lagen Werte in Form einer grafischen Darstellung vor. Aus dieser wurden die einzelnen Werte extrahiert und in Tabellenform überführt. Um den Verlauf möglichst gut wiederzugeben, wurde die folgende kubische Funktion angefittet:

$$Q = 2{,}19h^3 + 10{,}00h^2 + 0{,}63h \qquad (7.1)$$

mit dem Wasserstand h in *m* über dem Bezugsniveau und dem Abfluss *Q* in *m³/s*.

7.1.2 Niederschlagsdaten

Niederschlagsdaten lagen von neun verschiedenen Stationen vor, von denen sieben innerhalb und zwei außerhalb in unmittelbarer Nähe des oberen Dragonja-Gebietes lagen. Die Stationen sind gut über das Gebiet verteilt, mit einer leichten Häufung im westlichen Teil (siehe Abbildung 7.2).

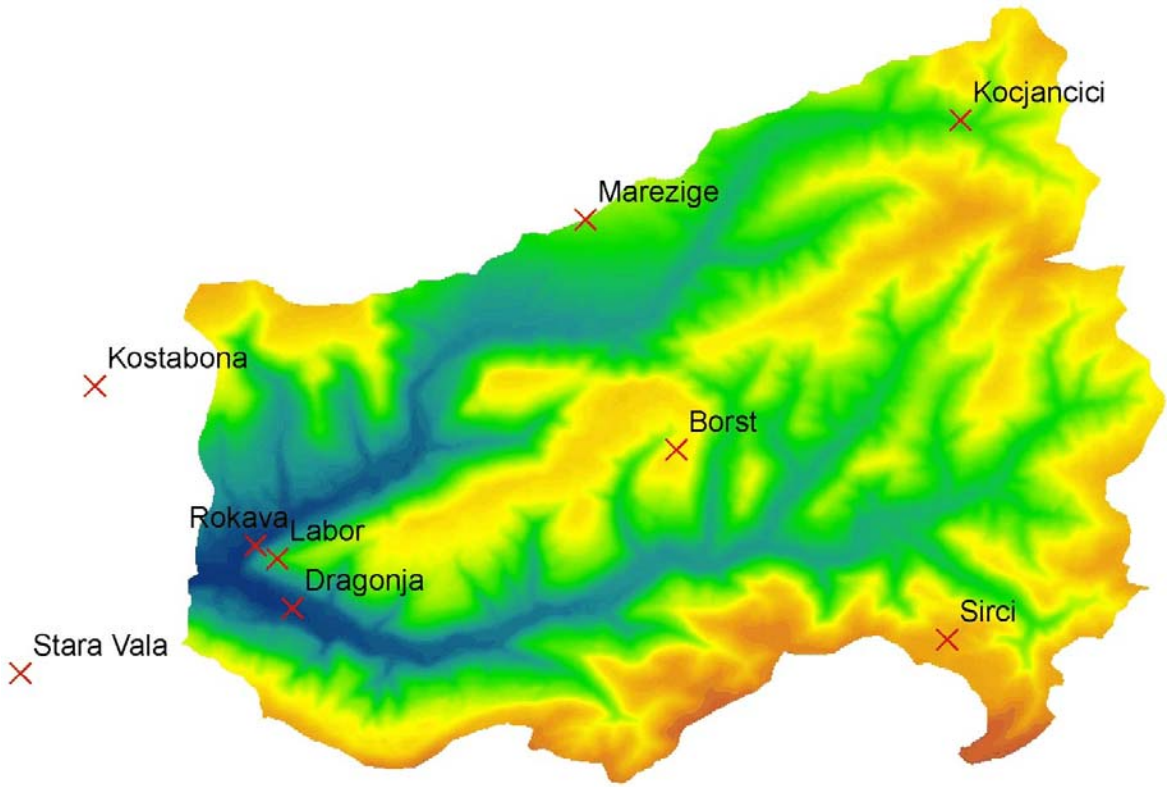


Abb. 7.2 Lage der Niederschlagsstationen, der Pegel und der Meteorologischen Station (Hintergrund: Darstellung des DEM)

Für die Station „Labor" waren keine Daten im Simulationszeitraum vorhanden. Von den anderen Stationen deckt keine den gesamten Modellierungszeitraum ab, es sind aber zu jedem Zeitpunkt Daten von mindestens drei Stationen vorhanden. Für die Zeiträume, in denen alle anderen notwendigen Daten vollständig waren, gab es Daten von mindestens fünf Stationen. Für die Berechnung der Niederschlags-Grids mit dem IDW-Verfahren wurden für jede Zelle nur die drei am nächsten liegenden Stationen berücksichtigt.

Die Niederschlagsdaten lagen als Logger-Dateien von Niederschlagswippen mit einem Umschlagvolumen von 0,1 mm vor. Als Zeitschritt für die ZIN-Simulationen wurden fünf Minuten gewählt, sodass auch der Niederschlag in einer Fünf-Minuten-Auflösung benötigt wurde. Dazu wurde ein Programm erstellt, das die Logger-Dateien zusammenfügt und entsprechend umwandelt, wobei zeitliche Lücken zwischen zwei Logger-Dateien als „noData" interpretiert werden.

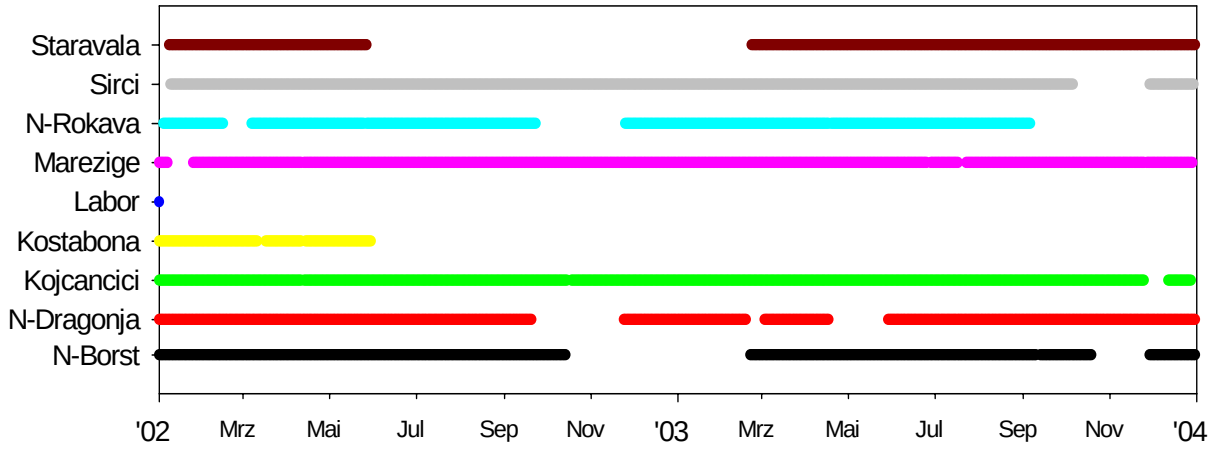


Abb. 7.3 Verfügbarkeit von Niederschlagsdaten der verschiedenen Stationen innerhalb der Simulationsperiode

7.1.3 Meteorologische Daten

Zur Modellierung der Verdunstung mit dem TRAIN-Modell werden neben den Niederschlagsdaten weitere meteorologische Daten benötigt:

- die Temperatur in ˚C,
- die relative Luftfeuchte,
- die Windgeschwindigkeit in m/s sowie
- die relative Sonnenscheindauer („SSD").

Für die Verteilung der vom TRAIN berechneten täglichen Verdunstung auf die ZIN-Modellierungszeitschritte wird vom ZIN-Modell außerdem die Globalstrahlung in W/m² benötigt. Diese Daten wurden im Dragonja-Gebiet an der meteorologischen Station „Borst" (siehe Abbildung 7.2) erhoben und lagen für die in Abbildung 7.4 gezeigten Zeiträume vor.

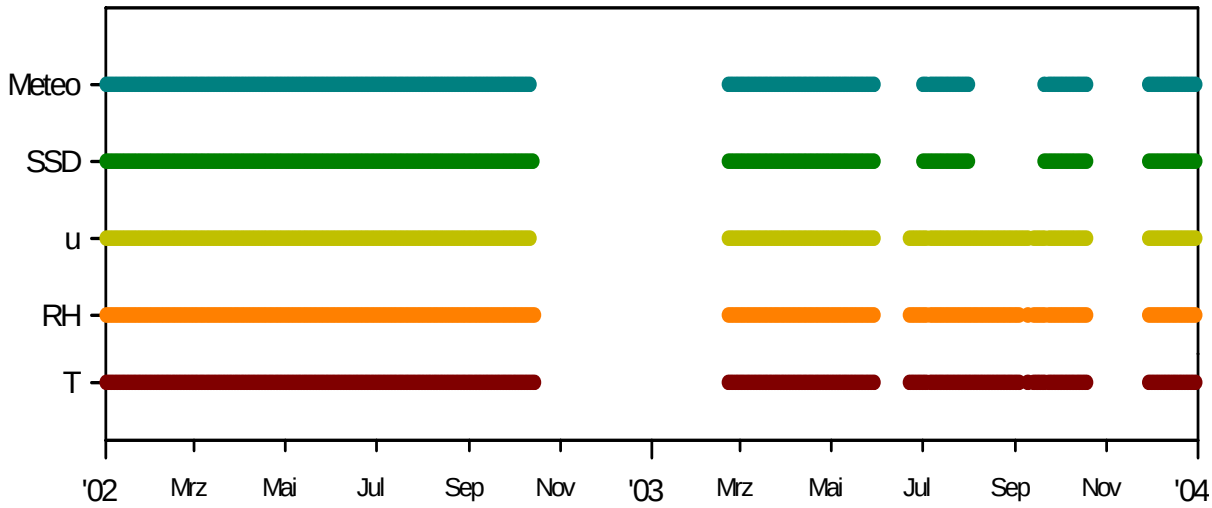


Abb. 7.4 **Verfügbarkeit meteorologischer Daten der Station Borst** (von unten nach oben: Temperatur, relative Luftfeuchte, Windgeschwindigkeit, relative Sonnenscheindauer, alle Daten gleichzeitig)

7.1.4 Räumlich verteilte Daten

Informationen über die Einzugsgebietshöhen lagen in Form digitaler Höhenlinien in 5 m-Abständen vor. Diese wurden in ArcGis in ein Raster-DEM mit 15 m Kantenlänge umgewandelt.

Die vom Grid mit den Landnutzungsklassen abgedeckte Fläche stimmte nicht überall mit den Einzugsgebietsgrenzen überein, einzelne Pixel fehlten am Rand und innerhalb des Gebietes. Diesen Pixeln wurde mit Hilfe des ArcGis-Werkzeuges „Euclidian Allocation" der Wert der nächsten definierten Zelle zugewiesen.

Die Verteilung der Landnutzung lag ebenfalls als Grid in 15 m-Auflösung vor. Sie umfasste die sieben Klassen

1. pasture
2. abandoned
3. arable
4. young forest
5. old forest

6. erosion
7. riverbed.

7.2 Parametrisierung

7.2.1 Einteilung der Teileinzugsgebiete

Das Gebiet wurde in 36 Teileinzugsgebiete für die Dragonja und weitere 15 Teilgebiete für die Rokava unterteilt (siehe Abbildung 7.5). Ein weiteres Gebiet enthält den Zusammenfluss dieser beiden Flüsse, unterhalb davon wurde ein weiteres Teilgebiet angefügt, um die Möglichkeit zu haben, auf ein Gerinnesegment mit dem vereinigten Abfluss aus beiden Gebieten zugreifen zu können.

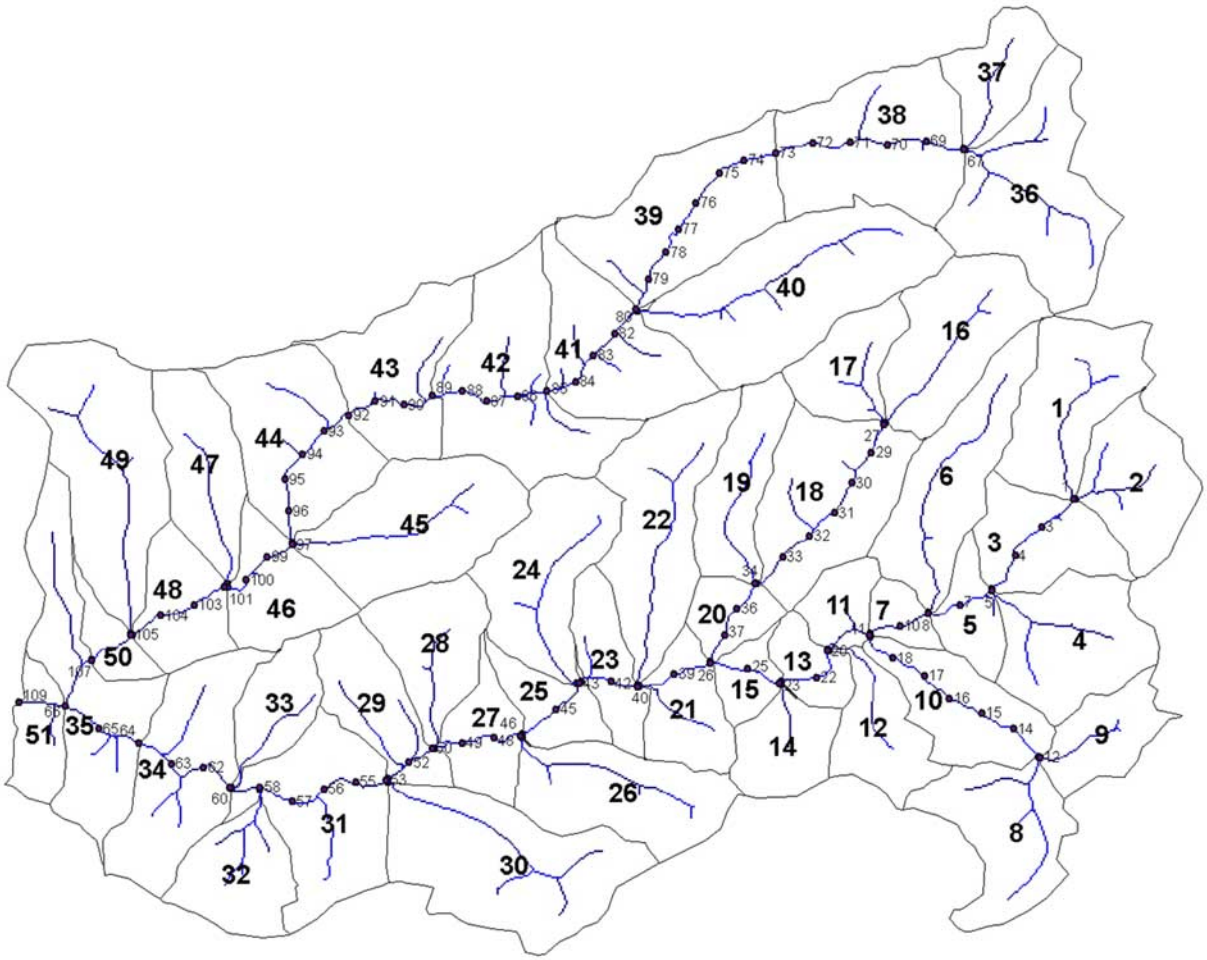


Abb. 7.5 Einteilung der Teileinzugsgebiete und der Gerinnesegmente

Die Einteilung erfolgte so, dass

1. die Pegel an einem Teilgebietsauslass lagen (Dragonja: Ezg 34, Segment 64 und Rokava: Ezg 50, Segment 107),
2. die Gebiete eine ähnliche Größe haben (im Mittel etwa 1km²)
3. die Zahl der Gebiete einen Kompromiss darstellt, was die Verlagerung der Abflusskonzentrations-Modellierung Richtung Routing und die Rechenzeit (siehe Kapitel 5.3.2) angeht, die mit steigender Zahl von Teileinzugsgebieten zunimmt.

7.2.2 Boden- und Abflussbildungsparameter für das ZIN-Modell

Zur Bestimmung der Bodenparameter für das ZIN-Modell lagen Messungen an mehreren Standorten vor, die alle in den Teileinzugsgebieten 34 und 48 zwischen Dragonja und Rokava (siehe Abbildung 7.5) lagen.

Über die Bodentiefen lagen Angaben nur als Tiefen von Probennahmen vor, in den meisten Fällen ohne die Angabe, ob dabei das anstehende Gestein erreicht wurde. Diese Tiefen wurden daher als Mindesttiefen interpretiert. An einer Stelle wurde die Mächtigkeit bis zum anstehenden Gestein mit 1,65 m angegeben.

Außerdem lagen für diese Standorte Korngrößenverteilungen vor, aus denen mit Hilfe des SPAW-Analysis-tools (SAXTON 2007) die effektive Porosität, der permanente Welkepunkt, die hydraulische Leitfähigkeit, sowie die Feldkapazität abgeschätzt wurden.

Der Porengrößenverteilungsindex λ nach Brooks-Corey wurde nach der bei RAWS ET AL. (1993) angegebenen empirischen Gleichung

$$\lambda = \exp\left[\begin{array}{l} -0,7842831 + 0,01775444(S) - 1,062498(\phi) - 0,00005304(S^2) - \\ 0,00273493(C^2) + 1,11134946(\phi^2) + 0,03088295(S)(\phi) + \\ 0,00026587(S^2)(\phi^2) - 0,00610522(C^2)(\phi^2) - \\ 0,00000235(S^2)(C) + 0,00798746(C^2)(\phi^2) - 0,00674491(\phi^2)(C) \end{array}\right] \qquad (7.2)$$

S: Sandanteil (%)

C: Tonanteil (%)

ϕ: Porosität (% Vol)

berechnet.

Über die räumliche Verteilung der Bodenparameter waren nur qualitative Informationen vorhanden. So hat der Boden nach GLOBEVNIK (1998) auf den Bergrücken eine Mächtigkeit von bis zu über einem Meter, ist auf den Hängen deutlich flachgründiger oder sogar ganz abwesend und in den Tälern bis zu mehreren Metern mächtig. Dementsprechend wurde für die Talböden ein Bodentyp mit einer Mächtigkeit von drei Metern angenommen. Für das übrige Gebiet wurde die Landnutzungskarte zur räumlichen Differenzierung benutzt.

Es wurde angenommen, dass das gesamte Gebiet zum Basisabfluss beiträgt.

7.2.3 Boden- und Landnutzungsparameter für das TRAIN-Modell

Die Boden- und Landnutzungsparameter für das Verdunstungsmodell TRAIN können vom Anwender nicht ohne Weiteres selbst festgelegt werden, sondern stehen im Quellcode des Programms. Der Anwender hat die Möglichkeit, aus den vorgegebenen Klassen die geeigneten auszuwählen. Für die Böden wurde aus Mangel an detaillierteren Informationen Lehmboden (Boden-Code 40) für das gesamte Gebiet angenommen.

Die ursprünglichen Landnutzungen wurden den im TRAIN vorhandenen Klassen wie in Tabelle 7.1 dargestellt zugewiesen.

Tabelle 7.1 Zuordnung der ursprünglichen Landnutzungsklassen zu TRAIN-Landnutzungsklassen

TRAIN-Code	TRAIN-Name	zugewiesene Landnutzungen
30	„Ackerland”	„arable”, „erosion”
60	„Wiesen, Weiden, Grünland”	„abandoned”, „pasture”
90	„Mischwald”	„young forest”, „old forest”
95	„Gewässer”	„riverbed”

Die einzige Anpassung im TRAIN-Quelltext betraf die Mächtigkeit der durchwurzelten Bodenschicht im Zuge der Modellkalibrierung. Für die TRAIN-Klasse „Grünland” („abandoned” und „pasture” im ursprünglichen Landnutzungsgrid) wurde die Wurzeltiefe von 900mm auf 20mm, für die TRAIN-Klasse „Ackerland” („arable” und „erosion” im ursprünglichen Landnutzungsgrid) von 1000mm auf 500mm reduziert.

7.2.4 Gerinneparameter

Die Gerinneparameter-Datei (Schlüsselwort „typeProps" in der Steuerungsdatei) enthält außer *Mannings n* und einem Formparameter *varper* ausschließlich für die Modellierung von Transmission-Losses relevante Werte. Da die Gerinneparameter abgeschätzt werden mussten, wurde darauf verzichtet, unterschiedliche Gerinnetypen zu differenzieren. Für *Mannings n* wurde ein einheitlicher Wert von 0,04 verwendet, für den Formfaktor *varper* ein Wert von 0,6. Für die die Überflutungscharakteristik bestimmenden Parameter *x* und *d* (beschrieben in LEISTERT (2005), Schlüsselworte „channel_x" bzw. „channel_d" in der Steuerungsdatei), wurde jeweils ein Wert von 1 angenommen.

Die Breiten werden in der Datei, die auch das Gerinnenetz enthält (Schlüsselwort „River-Net"), für jedes Segment angegeben. Sie wurden aus über die Software „Google Earth" (GOOGLE INC. 2007) erhältlichen Luftbildern abgeschätzt. Sohlgefälle und Länge der einzelnen Segmente stehen in derselben Datei und wurden aus der Höhendifferenz des DEM bzw. der Differenz des Fließweges zum Gebietsauslass vom Anfang zum Ende jeden Segments bestimmt. Die „RiverNet"-Datei findet sich im Anhang B.

7.3 Fazit

Die acht gut verteilten Niederschlags-Stationen sind für eine Anwendung des IDW-Verfahrens gut geeignet. Trotz vorhandener Datenlücken stehen zu jedem Zeitpunkt mindestens fünf Stationen zur Verfügung.

Bei den Bodenparametern wären detailliertere Daten wünschenswert gewesen. Die räumliche Verteilung unterschiedlicher Böden oder Bodeneigenschaften war weitgehend unbekannt und wurde für das ZIN-Modell über die Landnutzung abgeschätzt, für das TRAIN-Modell als homogen angenommen. Besonders Bodentiefe, hydraulische Leitfähigkeit, Infiltrationsrate und Porosität haben im ZIN-Modell einen großen Einfluss auf die Abflussbildung. Die Verwendung von Messdaten anstelle der Kalibrierung dieser Parameter würde daher dem physikalisch basierten Ansatz des Modells besser gerecht.

Die Werte einzelner für beide Modelle verwendeten Parameter sind teilweise unterschiedlich, was sich in der jetzigen Modellversion nicht vermeiden lässt, da die Parameter im TRAIN-Modell nur eingeschränkt verändert werden können.

8 Ergebnisse der Modellierung

Für die Kalibrierung des Modells wurde der Zeitraum vom 5.2.2002 bis zum 24.8.2002 verwendet, da hier sowohl alle zur Modellierung nötigen Daten als auch Abflussdaten vorhanden waren. Einen Überblick über die gesamte Zeit gibt Abbildung 8.1, anschließend werden die Ergebnisse noch einmal in kleineren Zeitblöcken dargestellt.

Für die Modellvalidierung wurde das Jahr 2003 verwendet. Auch hier waren ausreichend Daten für eine längere Modellierung vorhanden. Da das letzte Niederschlagsereignis, bei dem der gemessene Abfluss eine Reaktion zeigt, allerdings schon am 12.4.03 stattfand, ist die Validierungsphase mit drei Wochen und drei Abflussereignissen sehr kurz.

In den Diagrammen zur Darstellung der Ergebnisse sind jeweils auf der linken Achse modellierter und gemessener Abfluss aufgetragen, auf der rechten Achse der Niederschlag. Beim Niederschlag ist die Intensität des mittleren Gebietsniederschlags I_p in jedem Modell-Zeitschritt dargestellt (mm/h, dunkelblau). Da bei dieser Darstellung nicht zwischen räumlich stark beschränkten Niederschlägen mit hoher Intensität und gleichmäßig verteilten Niederschlägen geringer Intensität unterschieden werden kann, ist zusätzlich die im Gebiet maximal aufgetretene Intensität I_{P_max} für jeden Zeitschritt aufgetragen (10 mm/h, türkis).

8.1 Der Nash-Sutcliffe-Index

Ein gebräuchliches Verfahren zur Bewertung der Qualität einer Modellierung ist der Nash-Sutcliffe-Index (NASH & SUTCLIFFE, 1970). Er stellt eine Beziehung zwischen der Differenz aus modelliertem und gemessenem Abfluss sowie der Differenz aus beobachtetem und über den Modellierungszeitraum gemitteltem Abfluss her (Gleichung 8.1).

$$R_{eff} = 1 - \frac{\sum_{\Delta t} (Q_{obs} - Q_{sim})^2}{\sum_{\Delta t} (Q_{obs} - \bar{Q}_{obs})^2} \qquad (8.1)$$

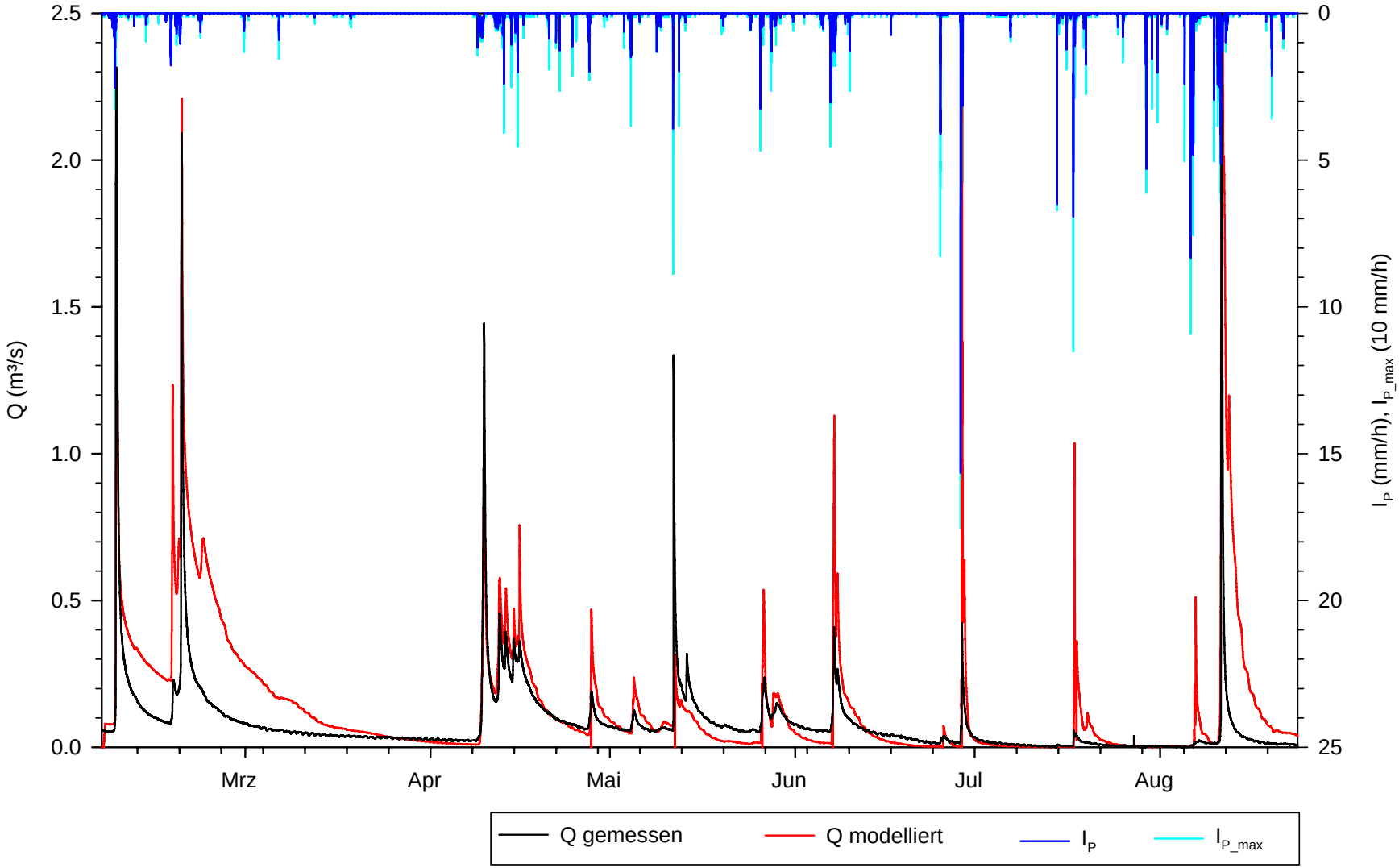


Abb. 8.1 Modellergebnis für den Kalibrierungszeitraum Februar bis August 2002

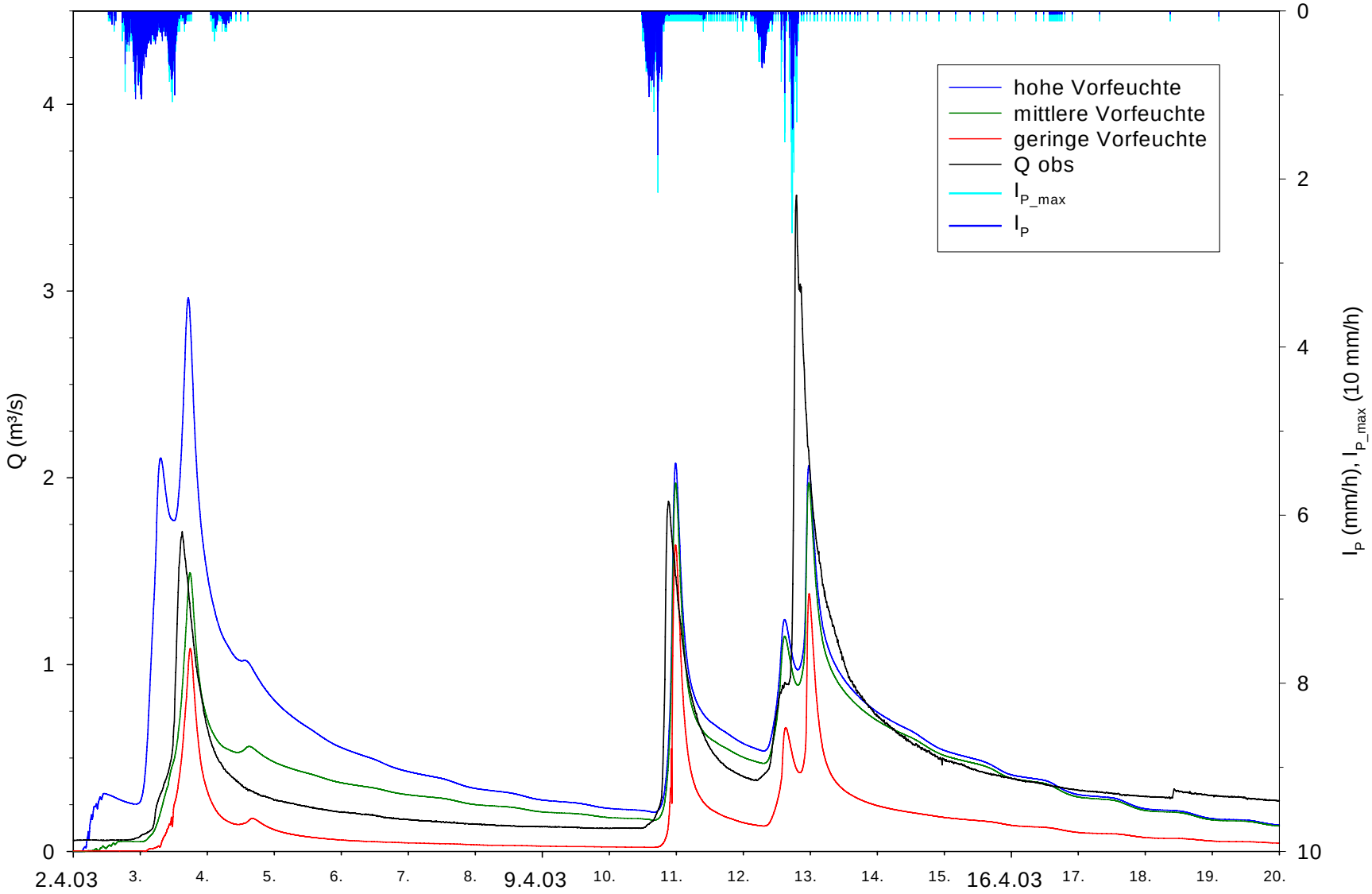


Abb. 8.2 Modellergebnis im Validierungszeitraum. Farbige Graphen: mit verschiedenen Vorfeuchten modellierter Abfluss

Der bestmögliche Wert für eine Modellierung, bei der das Ergebnis genau dem gemessenen Abfluss gleicht, liegt bei eins. Für eine Modellierung, bei der die Differenz zwischen modelliertem und gemessenem Abfluss genau so groß ist wie die zwischen gemessenem und durchschnittlichem Abfluss, ergibt sich ein R_{eff} von 0. Darunter liegende Werte zeigen ein Modellergebnis, dass stärker als der Mittelwert des Abflusses vom gemessenen Abfluss abweicht. Eine untere Grenze existiert dabei nicht.

8.2 Kalibrierungsphase

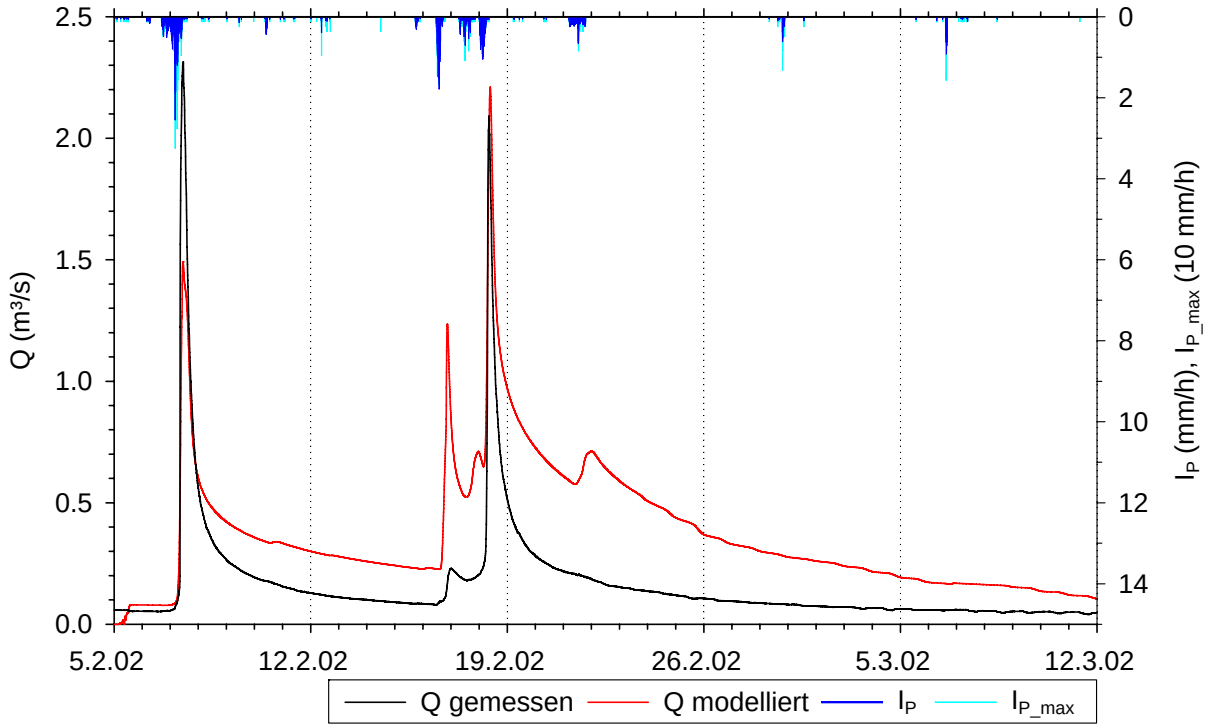


Abb. 8.3 Modellergebnisse Januar bis März 2002 (Modellstartphase)

Eine Berechnung der Modell-Effizienz nach Nash-Sutcliffe für die gesamte Kalibrierungs-Periode ergibt ein R_{eff} von -0,44. Das Ergebnis ist damit schlechter als der über den Modellierungszeitraum gemittelte gemessene Abfluss.

Der Vergleich von beobachtetem und modelliertem Abfluss ergibt ein uneinheitliches Bild. Während einige Ereignisse gut wiedergegeben wurden, wurden andere stark unter-, wiederum andere stark überschätzt. Dabei lassen sich drei Phasen unterteilen:

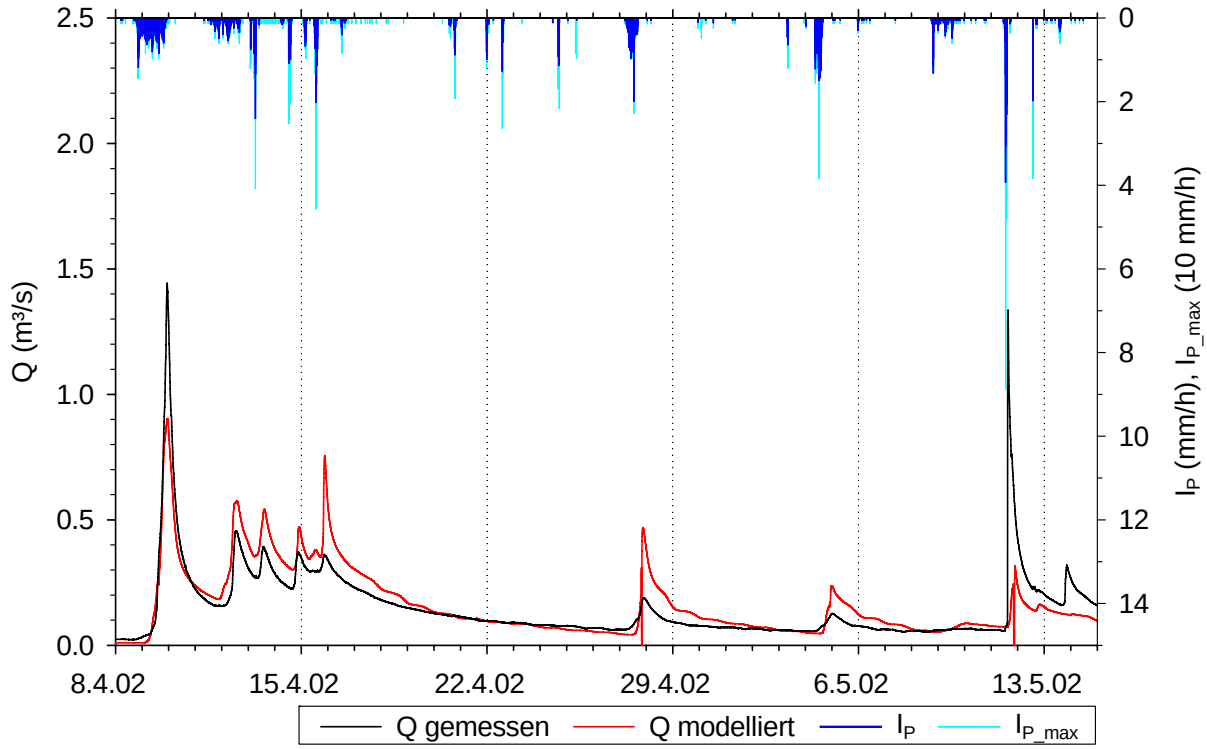


Abb. 8.4 Modellergebnisse April bis Mai 2002

- zwei Ereignisse nach dem Modellstart am 5.2.02, die der Warmlaufphase zugerechnet werden (siehe Abschnitt 8.4)
- April und Mai, in denen die Ereignisse größtenteils sehr gut wiedergegeben werden. Dabei werden Ereignisse bis etwa 0,5 m³/s überschätzt, darüber liegende teilweise deutlich unterschätzt
- ab Juni bis zum Ende der Modellierung am 23.8.02 werden durchweg alle Ereignisse überschätzt, teilweise um mehr als eine Größenordnung.

In Abbildung 8.3 ist das Ergebnis für die ersten fünf Wochen nach dem Modellstart am 5.2.02 dargestellt. Der deutlich zu hohe Basisabfluss ist möglicherweise ein Hinweis darauf, dass das Modell mit einer zu hohen Vorfeuchte gestartet wurde und sich noch in einer Warmlaufphase befindet. Auf eine weitere Analyse wird deshalb verzichtet.

Die nächste Periode vom 8.4.02 bis zum 14.5.02 (Abbildung 8.4) enthält die besten Modellergebnisse in der Kalibrierungsphase. Berechnet man die Modelleffizienz für den Monat April, ergibt dies ein R_{eff} von 0,80. Wird die anfängliche Niedrigwasserperiode ausgeschlossen, erhält man ein R_{eff} von 0,75 für die Zeit vom 9.4.02 bis zum 30.4.02.

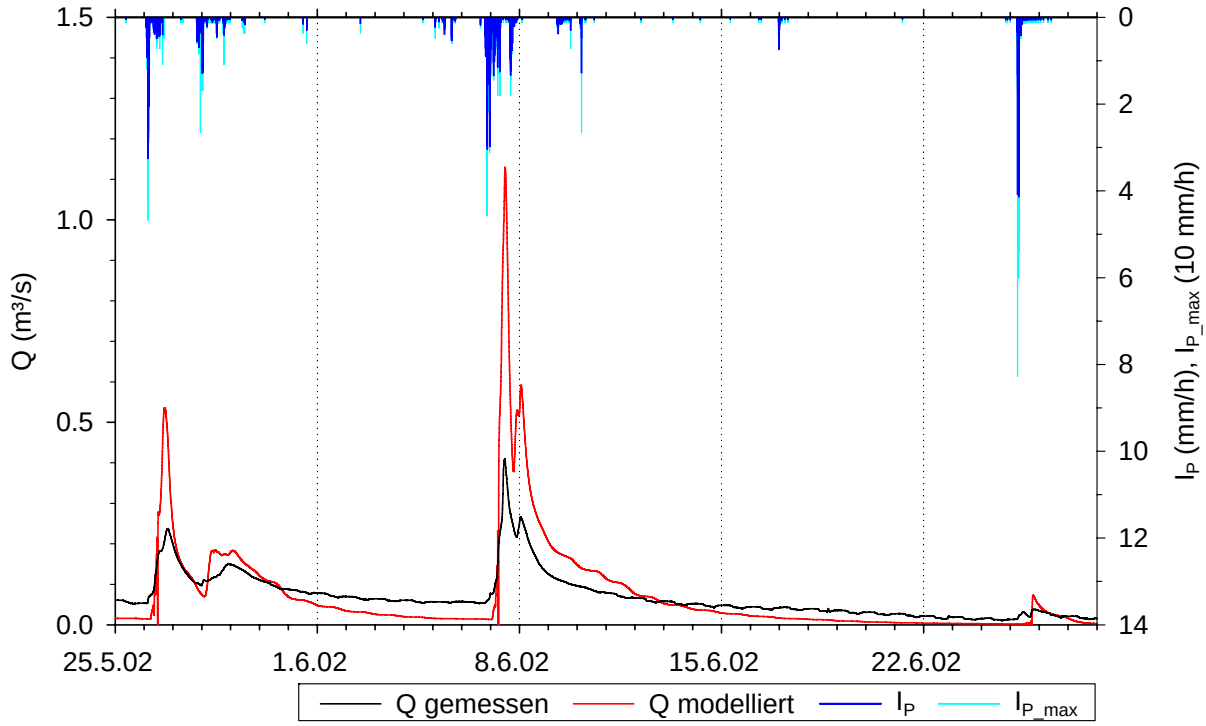


Abb. 8.5 Modellergebnisse Mai bis Juni 2002

Die beiden Ereignisse vom 9.4.02 und vom 11.5.02 sind deutlich unterschätzt worden, das erste mit 0,894 von 1,435 m³/s (62 %), das zweite sogar mit 0,308 von 1,315 m³/s (23 %). Dabei unterscheidet sich die Niederschlagscharakteristik sowie die Vorfeuchte. Dem ersten Ereignis ging eine trockene Periode von etwa einem Monat voraus (siehe Abbildung 8.1), das zugehörige Niederschlagsereignis war von geringer Intensität (maximal 14 mm/h) und etwas über einem Tag Dauer. Das Ansteigen und Abfallen einen Tag um den Abflusspeak herum wird sehr gut wiedergegeben, die Abflussspitze ist jedoch zu gering.

Das zweite Ereignis (11.5.02) folgt auf eine Periode mit deutlich mehr Niederschlag und höherem Basisabfluss, was auf eine höhere Vorfeuchte hindeutet. Hier ist der Niederschlag nur von kurzer Dauer (zwei kurze Ereignisse innerhalb von 2,5h) aber hoher Intensität (bis 89 mm/h). Der modellierte Abflusspeak erreicht hier nur etwa 23 % des gemessen Abflusses. Auch das Abflussvolumen ist erkennbar unterschätzt worden.

Ein Vergleich der Niederschlagsmenge bei diesem Ereignis von 1,6 mm (mittlerer Gebietsniederschlag) mit der Menge von 3,6 mm, die bei einem weiteren Ereignis am 4.5.02 zu einem weitaus kleineren Abfluss geführt hat, legt die Vermutung nahe, dass die Niederschlagsintensität

für die Abflussbildung am 11.5.02 eine wichtige Rolle gespielt hat. Ein weiterer Vergleich mit dem Ereignis vom 28.6.02 (Abbildung 8.6), bei dem nach einer langen trockenen Periode hohe Niederschlagsintensitäten bis zu 175 mm/h bei einer Gesamtmenge von 5,7 mm auftraten und zu einem Spitzenabfluss von lediglich 0,423 m³/s führten, zeigt allerdings, dass eine isolierte Betrachtung der Niederschlagsintensität für die Modellierung der Abflussbildung in diesem Gebiet nicht zu befriedigenden Modellergebnissen führen kann.

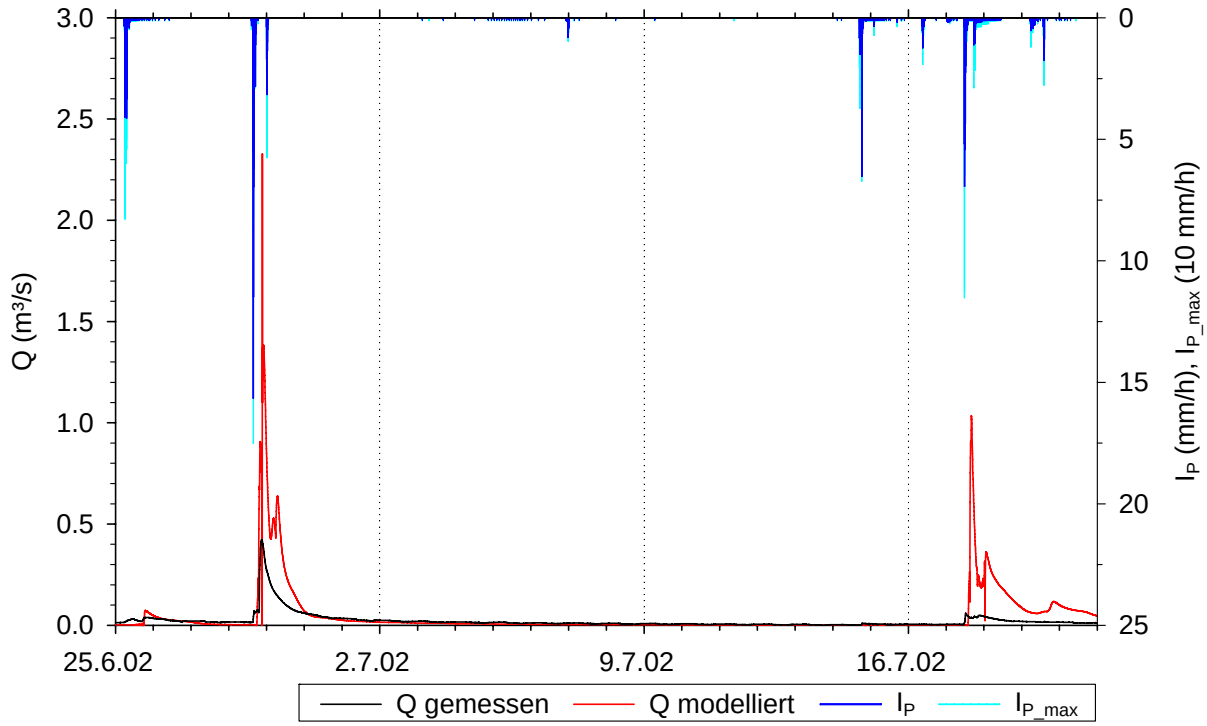


Abb. 8.6 Modellergebnisse Juni bis Juli 2002

Im derzeitigen Entwicklungsstand des ZIN-Modells findet eine Berücksichtigung der Niederschlagsintensität für die Abflussbildung über eine konstante Infiltrationsrate statt. Dies führte zu einem Dilemma bei der Kalibrierung. Wurden als realistisch erachtete Infiltrationsraten verwendet, wurden Abflüsse im Sommer mit ihren Niederschlägen hoher Intensität und kurzer Dauer noch deutlich stärker überschätzt als im endgültigen Setup. Daher wurden die Infiltrationsraten hochgesetzt, teilweise bis an die Grenzen des noch realistisch erscheinenden. Dadurch verminderte sich allerdings der Einfluss der Niederschlagsintensität auf die Abflussbildung, was teilweise zu einer drastischen Unterschätzung von Abflüssen führte, wie am Beispiel des 11.5.02 beschrieben wurde.

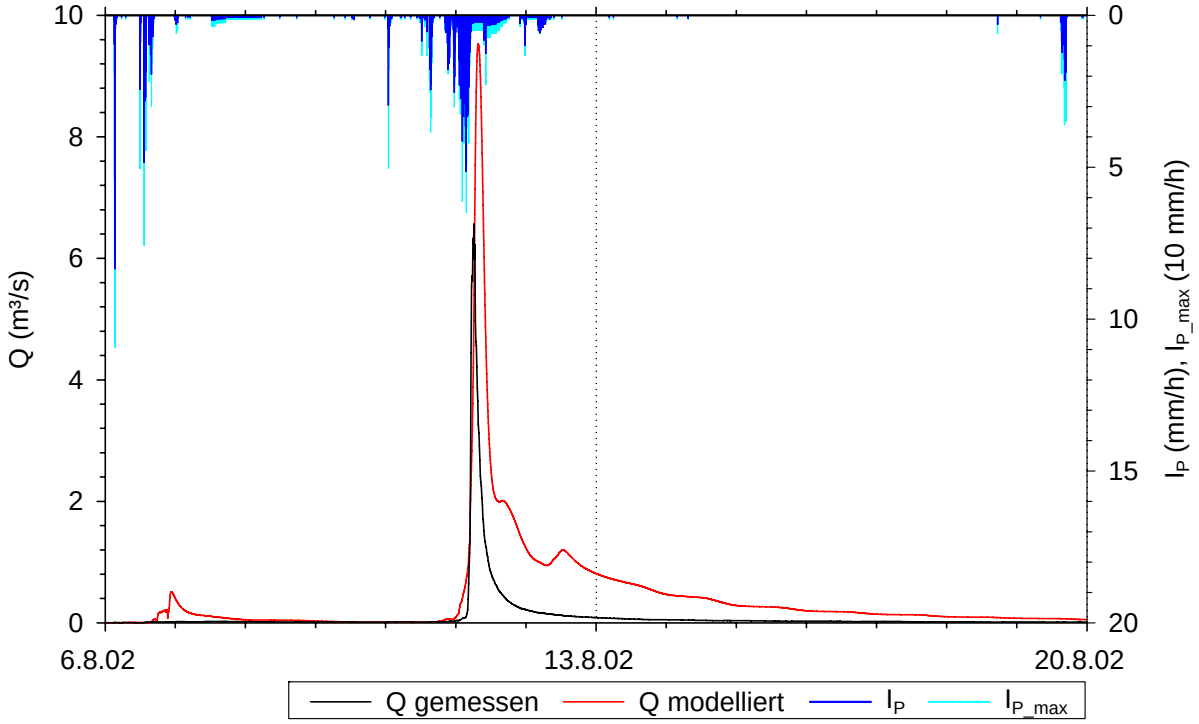


Abb. 8.7 Modellergebnisse August 2002

8.2.1 Verhalten des Routing

Die modellierte Abflussganglinie zeigt im Allgemeinen einen für Abflussganglinien realistisch erscheinenden Verlauf. Ein Oszillieren, wie es in früheren Anwendungen zu beobachten war, bzw. das durch Bildung des gleitenden Mittels eliminiert wurde, ist nicht mehr zu beobachten. Auch tägliche Einbrüche des Abflusses um Mitternacht treten nicht mehr auf. Allerdings zeigt der Verlauf gelegentlich während Abflussereignissen ein unrealistisches Verhalten, indem er kurzzeitig einbricht, um sofort danach wieder anzusteigen. Dies geschieht besonders deutlich während der Ereignisse vom 27.4.02 und 11.5.02 (Abbildung 8.4), 26.5.02 und 7.6.02 (Abbildung 8.5) sowie 28.6.02 und 17.-18.7.02 (Abbildung 8.6). Allen diesen Ereignissen ist gemeinsam, dass der modellierte Abfluss einen sehr steilen Anstieg zeigt. Vermutlich äußert sich hier die in Abschnitt 4.1.5.3 beschriebene Schwäche des Muskingum-Cunge-Verfahrens bei der Wiedergabe steil ansteigender Abflusskurven.

8.3 Validierungsphase

Die Validierungsphase umfasste den Zeitraum 2.4.03 - 22.4.03. Die Ergebnisse sind in Abbildung 8.2 dargestellt. Dabei wurden drei Modellläufe mit unterschiedlichen Bodenfeuchten durchgeführt. Eine der verwendeten Bodenfeuchten erbrachte einen deutlich zu niedrigen, eine einen deutlich zu hohen Abfluss bezogen auf den gemessenen Basisabfluss vor dem Ereignis. Bei einem Lauf zeigte der modellierte Basisabfluss eine gute Übereinstimmung mit dem gemessenen Abfluss.

Die Modell-Effizienz nach Nash-Sutcliffe lag für den Modelllauf mit zu geringer Vorfeuchte bei R_{eff} = 0,10 und für denjenigen mit zu hoher Vorfeuchte bei R_{eff} = 0,09. Für den Lauf mit angepasster Bodenfeuchte lag der Wert mit R_{ef} = 0,63 deutlich höher.

Dass die Modell-Effizienz bei allen drei Versionen höher liegt als in der Kalibrierungsphase, entspricht zunächst nicht den Erwartungen. Allerdings sind die Vorbedingungen kaum vergleichbar, da eine Anpassung der Vorfeuchte in der Kalibrierungsphase nicht vorgenommen wurde.

8.4 Warmlaufphase des Modells

8.4.1 Anpassung der Bodenfeuchte

Die Feuchte einer Bodenzelle ist ein wichtiger Faktor in der Berechnung der ungesättigten hydraulischen Leitfähigkeit nach van Genuchten im Bodenspeicher-Teil des Modells. Die daraus berechnete Sickerung aus der Bodenzelle heraus macht im vorliegenden Modell-Setup den größten Teil des Abflusses aus. Damit ergibt sich die große Bedeutung der Vorfeuchte, die angenommen wird, für die Ergebnisse der Anfangsphase eines Modelllaufs und die Frage, nach welcher Zeit sich die Ergebnisse von Modellierungen einander annähern, die mit unterschiedlicher Vorfeuchte bei ansonsten gleichen Parametern durchgeführt werden.

Für die Validierungsphase im April 2004 (siehe Abbildung 8.2) wurde ursprünglich eine sehr geringe Vorfeuchte verwendet, die am ersten Tag der Modellierung einen Basisabfluss von etwa 2,5 l/s brachte, während der gemessene Abfluss bei ca. 59 l/s lag. Ein folgender Modelllauf mit höherer Vorfeuchte erbrachte einen deutlich zu hohen anfänglichen Basisabfluss von etwa 280 l/s. Ein weiterer Versuch traf den Basisabfluss schließlich mit 53 l/s recht gut. In Abbildung 8.8 ist jeweils die Differenz zwischen den Modellergebnissen mit ungünstiger Vorfeuchte und dem Ergebnis mit passender Vorfeuchte dargestellt.

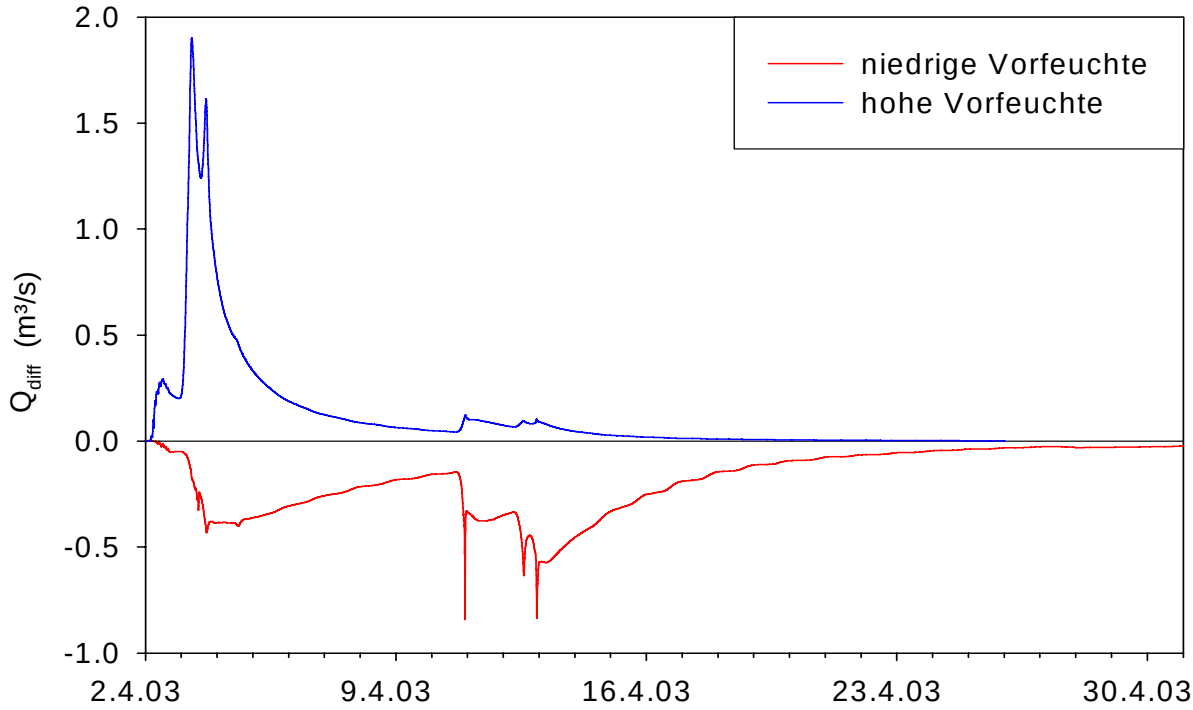


Abb. 8.8 Verlauf der Angleichung drei verschiedener Modellläufe mit unterschiedlichen Vorfeuchten

Es zeigte sich, dass sich die mit höherer Vorfeuchte erhaltenen Ergebnisse deutlich schneller anpassten, als die mit niedrigerer Vorfeuchte. Am 20.4.03, also 19 Tage nach Simulationsbeginn, betrug die Differenz nur noch 5 l/s.

Bei den Ergebnissen mit geringerer Vorfeuchte war nach einem Monat am 2.5.03 immer noch eine Differenz von 17 l/s vorhanden. Auffällig ist außerdem, dass die Differenz nach den Niederschlagsereignissen vom 10.4. und 12.4. sogar größer ist als am Anfang. Dies hängt vermutlich mit dem Verlauf der verwendeten Beziehung zwischen Bodenfeuchte und gesättigter Leitfähigkeit nach van Genuchten zusammen. Steigt die Feuchte um einen bestimmten Betrag an, steigt bei geringer Vorfeuchte die Sickerung deutlich langsamer, als bei höherer Vorfeuchte. Es ist also anzunehmen, dass sich die Bodenfeuchten gerade in einer Niederschlagsphase stärker annähern als in einer trockenen Phase, da der Abfluss in dem Modell-Setup mit höherer Vorfeuchte überproportional zunimmt und sich der Speicher somit weniger stark auffüllt, während beim Setup mit geringerer Feuchte ein größerer Teil des Niederschlags zur Auffüllung des Bodenspeichers beiträgt.

Die Länge einer Warmlaufphase hängt also wesentlich davon ab, wie gut die Vorfeuchte geschätzt werden kann. Wird die Vorfeuchte anhand des Abflusses bemessen, scheint eine Feuchte, die einen zu hohen Abfluss generiert, eine kürzere Warmlaufphase zu bedingen, als eine Vorfeuchte, die einen entsprechend niedrigeren Abfluss produziert.

Um zu einer zuverlässigen Abschätzung der nötigen Länge einer Warmlaufphase zu kommen, müssten weitere Tests durchgeführt werden, eventuell auch unter Berücksichtigung der räumlichen Verteilung der Feuchte.

8.4.2 Stabilisierung des Channel-Routing

Das Channel-Routing erhält seinen Input aus der Abflusskonzentration. Dieser Input setzt am ersten Tag der Simulation mit einer gewissen Verzögerung ein. Außerdem sind die Modell-Gerinne vollständig trocken, sodass hier eine gewisse Warmlaufzeit erwartet werden kann. Im Verlauf der Modellierungsarbeiten zeigte sich außerdem, dass das Channel-Routing anfangs nicht stabil läuft. Dies ist möglicherweise auf die bei trockenem Gerinne große relative Änderung des Abflusses schon bei geringen Zuflüssen zurückzuführen. Ein Vorlauf von einem Tag zur Berücksichtigung der Verzögerung des Inputs und der Stabilisierung des Routing wurde als nötig und ausreichend angesehen.

8.5 Trockenwetterabfluss

Das Ergebnis der Modellierung für eine trockene Periode im Februar bis März 2002 ist in Abbildung 8.9 dargestellt. Obwohl der modellierte Abfluss am Ende des ersten Tages ohne Niederschlag (22.2.02) mit 0,589 m³/s 3,8-mal so hoch ist wie der gemessene Abfluss (0,153 m³/s), ist er schon am 26.3.02 niedriger als dieser und fällt weiterhin deutlich ab, während der gemessene Abfluss zu diesem Zeitpunkt nur noch sehr wenig abnimmt.

Die Modell-Wasserbilanz für den Zeitraum vom 15.3.02 bis zum 7.4.02 zeigt, dass die Speicherentleerung dabei nicht wesentlich durch Abfluss, sondern durch Verdunstung erfolgt. Einer Speicheränderung von -40 mm stehen 2,6 mm Abfluss und 38 mm Verdunstung gegenüber (Niederschlag: 0,5 mm).

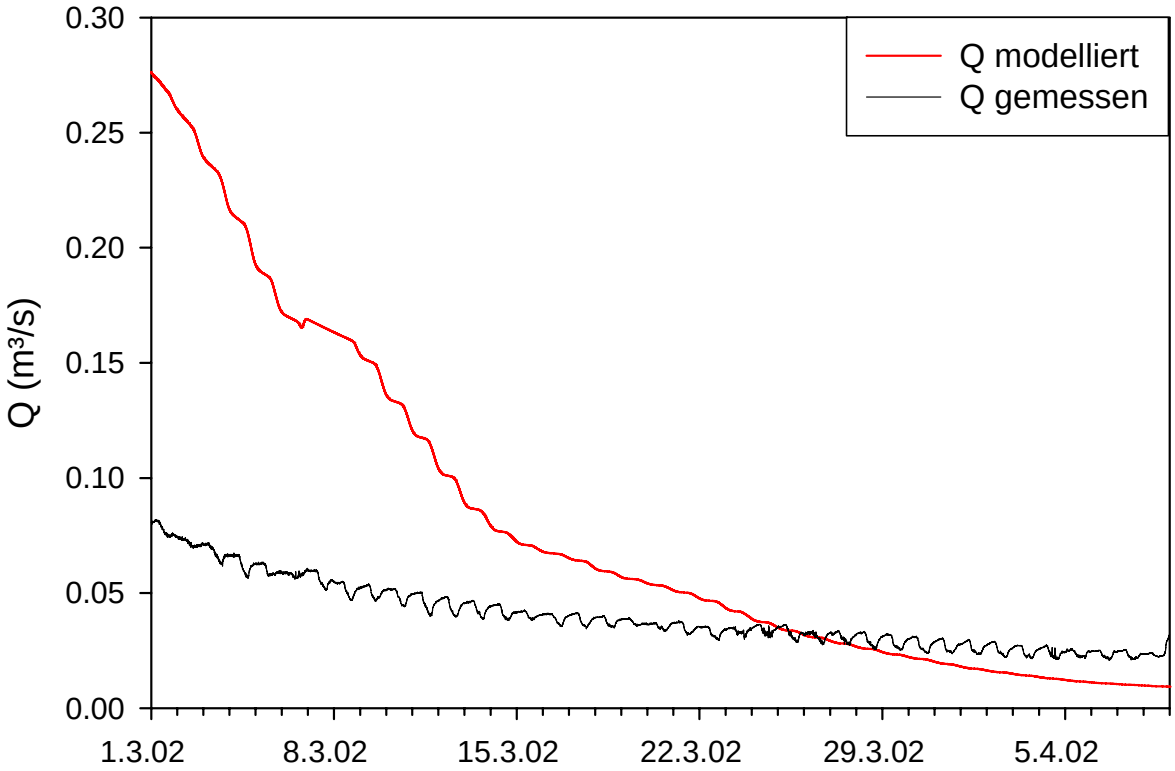


Abb. 8.9 Modellergebnis bei Trockenwetter im Februar/März 2002

Mögliche Erklärungen sind entweder ein zu kleiner Speicher oder eine Überschätzung der Verdunstung. Größere Bodentiefen als zusätzliche Speicher wären denkbar, allerdings wurde die Tiefe des Bodens im Wald mit 2 m und in den Talböden mit 3 m schon recht hoch angesetzt. Alternativ könnte ein Grundwasservorkommen vermutet werden, welches den Basisabfluss speist. Zurzeit bietet das Modell allerdings keine Möglichkeit, einen solchen Speicher darzustellen.

8.5.1 Tagesgang des Abflusses

In der gemessenen Abflussganglinie ist in Phasen mit sehr wenig oder ohne Niederschlag ein Tagesgang erkennbar, so beispielsweise im März 2002.

In Abbildung 8.10 ist die gemessene Abflussganglinie Q aufgetragen sowie der gleitende Mittelwert $\overline{Q}$ über einen Tag. Aus der Differenz dieser beiden Abflüsse ergibt sich der um den langfristigen Trend bereinigte Differenz-Abfluss Q_{diff}, der nur die tageszeitliche Schwankung

darstellt. Diese betrug im Mittel 6,7 l/s (3,7 - 10,9 l/s), wobei mit abnehmendem Abfluss auch die absoluten Schwankungen geringer wurden. Ab dem Abend steigt der Abfluss an, bis am Morgen die Strahlung und damit vermehrte Verdunstung einsetzt und der Abfluss wieder abfällt.

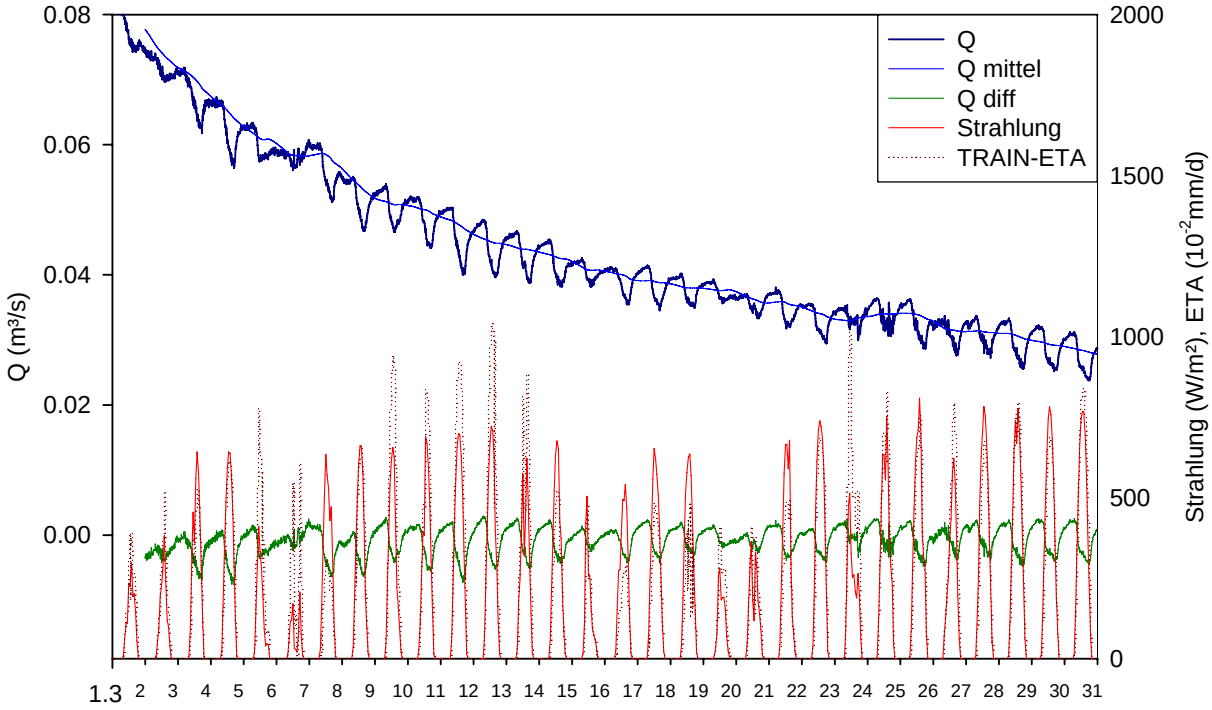


Abb. 8.10 Beziehung zwischen Strahlung, modellierter Verdunstung und Abfluss (März 2002)

An Tagen mit großer Strahlung ist dieser Zyklus besonders stark ausgeprägt, während er an Tagen mit schwacher Strahlung wie z. B. dem 6.3.02 fast gar nicht auftritt, bzw. wie am 19.3. und 20.3. deutlich schwächer ist.

Eine Wiedergabe des Anstiegs des Abflusses ist vom TRAIN-ZIN-Modell nicht zu erwarten, da der modellierte Basisabfluss nur durch ein Niederschlagsereignis und damit eine Erhöhung der Speicherfüllung ansteigen kann, nicht aber durch eine Abnahme der Evapotranspiration. Im modellierten Abfluss ist allerdings ein treppenförmiges Abfallen des Abflusses in trockenen Perioden zu beobachten, dessen Ausprägung ebenfalls von der Strahlungsintensität abhängig ist. Das Abfallen des Abflusses endet hier allerdings erst kurz nach Mitternacht, während das verstärkte Abfallen erst am Nachmittag beginnt. Während der gemessene Abfluss dem Strahlungsverlauf ohne nennenswerte Verzögerung folgt, weist die Reaktion des modellierten

Abflusses eine Verzögerung im Bereich von etwa sieben Stunden auf und kann damit den tatsächlichen Verlauf des Abflusses auch zeitlich nicht genau wiedergeben. Allerdings kann auch die Wiedergabe der Geschwindigkeit dieser Reaktion nicht vom Modell erwartet werden. Die Darstellung der Abflusskonzentration erfolgt sowohl für Oberflächenabfluss als auch für den Basisabfluss mit dem Unit-Hydrograph-Ansatz wie in Kapitel 5.3.2 beschrieben. Als mittlere Konzentrationszeit für ein Teileinzugsgebiet wurde dabei eine Zeit von 200 Minuten verwendet. Wird zur groben Abschätzung der zusätzlichen Verzögerung durch die Fließzeit im Gerinne eine Fließgeschwindigkeit von 0,3 m/s und eine Fließstrecke von 1/3 der Gesamtlänge des Hauptgerinnes (ca. 11km) angenommen, ergibt sich eine Gesamtverzögerung von 6,75 h, was gut zu der im Modell beobachteten Verzögerung passt.

Der deutliche Unterschied zwischen beobachtetem und gemessenem Tagesgang für Trockenperioden lässt darauf schließen, dass das Unit-Hydrograph-Verfahren in der verwendeten Form nicht für eine präzise Wiedergabe des Abflusses im Niedrigwasserbereich geeignet ist. Da die zeitliche Änderung des Abflusses bei Niedrigwasser allerdings recht langsam ist, führt auch ein großer zeitlicher Fehler nur zu einem geringen Fehler in der Modellierung der Abflusshöhe.

8.6 Fazit

Die Modellierung des Dragonja-Gebietes erfolgte in einer gut sechsmonatigen Kalibrierungs- und einer mit drei Wochen sehr kurzen Validierungsphase. Eine Berechnung der Modelleffizienz nach Nash-Sutcliffe ergab für den Kalibrierungszeitraum ein R_{eff} von -0,44.

Vergleiche zwischen Ereignissen mit verschiedenen Vorfeuchten, Niederschlagsmengen und Niederschlagsintensitäten legen den Schluss nahe, dass bei der Abflussbildung im Dragonja-Gebiet eine Abhängigkeit zumindest von Vorfeuchte, Niederschlagsmenge und Niederschlagsintensität besteht, wobei nicht einzelne dieser Faktoren, sondern ihre jeweilige Kombination entscheidend ist.

Die Form der Abflussganglinien lässt darauf schließen, dass frühere Probleme des Channel-Routings bezüglich der Stabilität und anderem unerwartetem Verhalten weitgehend beseitigt werden konnten. Weiterhin zu beobachtende Einbrüche der Ganglinie vor steilen Anstiegen sind vermutlich verfahrensbedingt.

Bei unbekannter Vorfeuchte muss eine Warmlaufphase eingeplant werden, in der sich die Ergebnisse von Läufen mit unterschiedlicher Vorfeuchte angleichen. Der Verlauf dieser Angleichung wurde beispielhaft für die Modellierung eines Ereignisses mit drei verschiedenen Vor-

feuchten dargestellt. Die Dauer der Angleichung der Modellergebnisse lag im Beispiel bei einigen Wochen. Dabei schien eine zu hohe anfängliche Vorfeuchte günstiger zu sein als eine zu niedrige. Auch der Abflusskonzentrationsteil und das Channel-Routing brauchen eine gewisse Vorlaufzeit, da sie anfänglich kein Wasser enthalten. Für diesen Vorlauf wird ein Tag als ausreichend angesehen.

In Trockenwetterperioden wurde im gemessenen Abfluss ein Tagesgang beobachtet. Durch Vergleiche der Ausprägung dieses Tagesganges mit der Strahlung konnte gezeigt werden, dass dieser auf die Verdunstung zurückzuführen ist. Auch im modellierten Abfluss ist eine tagesperiodische Treppenform zu erkennen, die allerdings etwa 7h verzögert auftritt und keinen nächtlichen Anstieg aufweist. Eine exakte Wiedergabe ist aufgrund der Modellkonzeption auch prinzipiell nicht möglich. Allerdings spielt selbst ein vergleichsweise großer zeitlicher Fehler bei geringen Abflussänderungen, wie sie bei Niedrigwasser auftreten, nur eine untergeordnete Rolle.

9 Diskussion und Ausblick

Mit Ausnahme der Transmission-Losses wurden alle Modellkomponenten unterschiedlich stark überarbeitet, teilweise nur nach programmiertechnischen Aspekten, teilweise auch inhaltlich. Die Verarbeitung von Niederschlags-Stationsdaten sowie das Benutzer-Interface wurden neu erstellt.

9.1 Verarbeitung von Niederschlagsdaten

Das Thiessen-Polygon-Verfahren und das IDW-Verfahren sind zwei bewährte Verfahren zur Übertragung von Punktdaten in die Fläche. Diese stehen nun im Modell zur Verfügung, sodass eine Aufbereitung von Stationsdaten zu Grids außerhalb des Modells, die bei großen Datenmengen schon aufgrund des Speicherbedarfs nicht praktikabel wäre, entfällt. Die Abwägung, welches der beiden Konzepte bzw. ob überhaupt eines davon für eine konkrete Anwendung geeignet ist, muss vom Anwender unter Berücksichtigung der bekannten Einschränkungen der Verfahren getroffen werden.

Sollte sich in einer künftigen Anwendung des Modells die Nutzung des IDW-Verfahrens als nicht zufriedenstellend erweisen, z. B. weil eine starke Clusterung der Stationen zu einer schlechten räumlichen Verteilung des Niederschlages führt, wäre eine Erweiterung um ein Verfahren wie z. B. Krigging, das diese Schwäche nicht aufweist, denkbar. Der Aufwand wäre vergleichsweise gering, da die grundlegende Struktur für die Verarbeitung von Stationsdaten nun geschaffen ist (Einlesen der Stationsdaten, Berechnung von Entfernungen, Schnittstelle zum restlichen Programm).

9.2 Abflusskonzentration

Der veränderte Unit-Hydrograph-Ansatz für die Abflusskonzentration beruht auf einer EV-I (Gumbel-) Verteilung, für die die Konzentrationszeit sowie ein Skalierungsfaktor angegeben werden kann. Zur Berücksichtigung der Unterschiedlichkeit der Teileinzugsgebiete wurde zusätzlich eine von Hangneigung und Größe des jeweiligen Einzugsgebietes abhängige automatische Anpassung dieser Parameter eingeführt. Während die Veränderung der beiden Ausgangsparameter eine gute Möglichkeit zur Feinabstimmung des Modells bezüglich des Eintretens von Abfluss-Peaks und der Auflösung dicht aufeinander folgender Abflussereignisse darstellt, hatte

die automatisierte Variation dieser Parameter für die Teileinzugsgebiete keinen erkennbaren Einfluss auf die Qualität der Modellergebnisse.

Bei der Anpassung des Konzentrationszeit-Parameters besteht insofern eine Beziehung zum Routing, als auch über die Gerinneparameter wie die Rauigkeit eine Anpassung des zeitlichen Verlaufs der Abflusswelle am Gebietsauslass möglich ist. Im Sinne einer physikalisch basierten Modellierung sollte das Augenmerk daher nicht ausschließlich auf dem für den Gebietsauslass modellierten Abfluss liegen, sondern auch soweit möglich eine realistische Abschätzung der Parameter für Abflusskonzentration und Routing aus Untersuchungen im jeweiligen Gebiet erfolgen.

9.3 Routing

Im Routing-Teil des Modells konnte durch einige programmtechnische Veränderungen sichergestellt werden, dass immer das nicht-lineare Muskingum-Cunge-Verfahren verwendet wird, während gleichzeitig die Terminierung der Iteration garantiert wird. In früheren Versionen bestehende Stabilitätsprobleme konnten teilweise beseitigt werden, lediglich durch das Verfahren bedingte Instabilitäten bei steil ansteigenden Abflusskurven bleiben bestehen.

9.4 Abflussbildung

Die Modelleffizienz bei der Modellierung der Dragonja war mit einem negativen R_{eff}-Wert von -0,44 nicht zufriedenstellend. Die Ursache ist vorwiegend in der Darstellung der Abflussbildung zu suchen. Die im Modell zur Verfügung stehenden Prozesse der Abflussbildung sind Infiltrationsüberschuss, Sättigungsflächenabfluss und aus der Sickerung generierter Abfluss. Infiltrationsüberschuss ist im Modell nicht von der Vorfeuchte abhängig. Die gemessenen Daten von Niederschlag und Abfluss weisen darauf hin, dass im Dragonja-Gebiet ein einfacher Zusammenhang zwischen Niederschlagsintensität und Abflussbildung ohne Berücksichtigung der Vorfeuchte nicht ausreicht, um die natürlichen Prozesse abzubilden. Der von der Feuchte abhängige Sättigungsflächenabfluss konnte dieses Problem nur teilweise kompensieren.

Letztendlich wurde der größte Teil des Abflusses über das neue „Baseflow"-Modul simuliert. Es scheint nicht verwunderlich, dass dieses nicht in der Lage ist, gleichzeitig den Ereignisabfluss und den Basisabfluss mit befriedigender Qualität darzustellen. Die Unterscheidung zwischen schnellen und langsamen Fließprozessen wurde im Zuge der Kalibrierung des Modells

eher über die unterschiedlichen Charakteristiken der verwendeten Bodentypen statt über unterschiedliche Prozesse erreicht, was allerdings eine Beschneidung des physikalisch basierten Ansatzes des Modells darstellt.

9.5 Mögliche Erweiterung des TRAIN-ZIN-Modells

Eine Weiterentwicklung der Abflussbildung im Modell durch eine Integration weiterer Abflussbildungsprozesse scheint essenziell für folgende Anwendungen in humiden Klimaten. Möglicherweise würde bereits die Einführung einer Vorfeuchte-Abhängigkeit der Infiltrationsraten, etwa nach dem für die Transmission-Losses im Channel-Routing bereits verwendeten Green & Ampt-Verfahren, zu einer deutlichen Verbesserung führen. Derart generierter Abfluss könnte u. U. nicht nur als Oberflächenabfluss, sondern auch als eine konzeptionelle Darstellung oberflächennahen Interflows interpretiert werden.

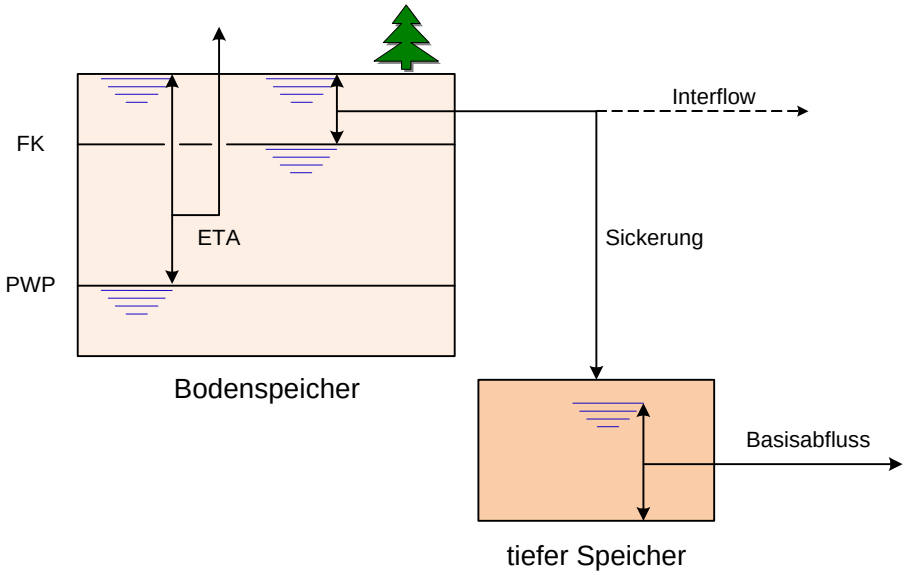


Abb. 9.1 Einbindung eines zusätzlichen Speicherelementes in das ZIN-Modell

Eine weitere Verbesserung der Repräsentation natürlicher Prozesse im Modell könnte auch durch die Einführung eines zusätzlichen Speichers erreicht werden. Denkbar wäre etwa die in Abbildung 9.1 gezeigte erweiterte Konzeption. Dabei findet Verdunstung aus dem oberen Bodenspeicher oberhalb des permanenten Welkepunktes statt. Sickerung tritt bis zum Erreichen

der Feldkapazität auf und speist einen tiefen Bodenspeicher (Grundwasserspeicher). Aus diesem wird ein füllungsabhängiger Beitrag zum Basisabfluss generiert.

9.6 Implementation des TRAIN-ZIN-Modells

Durch die Implementation eines Benutzer-Interfaces konnte die Anwenderfreundlichkeit des Modells erheblich verbessert werden. Durch die einfache Struktur des Interface sowohl auf der Anwender- als auch auf der Programmseite und den Verzicht auf ein komplexes GUI ist eine spätere Erweiterung um Steuerungselemente (Schlüssel-Werte-Paare) ohne Weiteres möglich. Die Struktur des Interface muss dazu an keiner Stelle geändert werden.

Eine weitere Geschwindigkeitsoptimierung des Modells wäre insbesondere bei Anwendungen von Vorteil, bei denen eine Kalibrierung stattfinden soll. Durch eine kürzere Programmlaufzeit könnte eine größere Zahl von Parameterkombinationen getestet werden, eventuell könnte sogar eine automatisierte Form der Parameterschätzung eingeführt werden.

9.6.1 Dokumentation

Ein wichtiger Faktor für die praktische Benutzbarkeit eines Programms ist die Existenz einer umfassenden Benutzer-Dokumentation. Aus dieser sollte ein Anwender erfahren können,

- welche Daten in welchem Format vorliegen müssen,
- welche Bedeutung die Eingabedaten haben und auf welche Weise sie im Programm verarbeitet werden,
- wie das Programm bedient werden muss.

Auch die Weiterentwicklung des Programms könnte durch eine eigene Dokumentation erheblich vereinfacht werden. Hilfreiche Bestandteile dieser Dokumentation wären zum Beispiel

- Programm-Strukturpläne auf verschiedenen Ebenen (gesamtes Programm, einzelne Klassen, ggf. noch feiner bei komplexem Quellcode),
- eine Darstellung der verwendeten Algorithmen und Datenstrukturen,
- eine Erläuterung des Bezugs zu den hydrologischen Konzepten: was ist wo und auf welche Art implementiert?
- eine Erklärung der wichtigen Variablennamen.

Der vielleicht wichtigste Punkt ist aber eine ausführlichen Kommentierung des Quelltextes, ohne die eine Einarbeitung in ein vorhandenes Programm fast unmöglich ist.

Trotz des zusätzlichen Aufwandes sollten die beiden genannten Formen der Dokumentation auch bei Weiterentwicklungen der Software mit gepflegt und auf dem aktuellen Stand gehalten werden.

9.7 Weitere Modellierung im Dragonja-Gebiet

Die in dieser Arbeit beschriebene Modellierung diente hauptsächlich dem Test des TRAIN-ZIN-Modells. Für eine weitere Modellierung im Dragonja-Gebiet wäre die bessere Repräsentation der Abflussbildungsprozesse im Modell wichtig, ebenso wie eine weitere Erhebung von Daten im Gebiet, insbesondere, was die Bodeneigenschaften betrifft.

Im Dragonja-Gebiet hat die Zahl der Einwohner seit 1945 abgenommen, durch die Aufgabe landwirtschaftlich genutzter Flächen hat gleichzeitig die Bewaldung zugenommen (KEESTRA 2007). Im Zuge dessen hat sowohl die Zahl der Tage mit sehr hohen als auch die mit sehr niedrigen Abflüssen abgenommen (GLOBEVNIK 1998). In anderen Gebieten wurde durchaus eine andere Wirkung des Waldes nachgewiesen (COSANDEY ET AL. 2005), sodass die weitere Entwicklung des Abflusses im Dragonja-Gebiet bei sich weiter ändernder Landnutzung ungewiss ist.

Interessant wäre daher die Modellierung verschiedener Landnutzungsszenarien. Neben einer Kalibrierung wäre auch die Validierung anhand von mehreren Sätzen historischer Landnutzungsdaten mit den dazugehörigen Abflussdaten wünschenswert. Damit könnte nicht nur die Gültigkeit des Modells für einen statischen Gebietszustand, sondern auch für den Übergang von einem Gebietszustand zu einem anderen überprüft werden.

10 Literatur

Breu J. (Hrsg.) (1970 - 1989): Atlas der Donauländer. Österreichisches Ost- und Südosteuropa-Institut, Wien.

Breymann U. (1999): C++. Eine Einführung. Hanser, München.

Chow V.T. (1964): Section 14: Runoff. Handbook Of Applied Hydrology. Editor: V.T. Chow. McGraw-Hill. New York

Cosandey C., Andréassian V., Martin C., Didon-Lescot J.F., Lavabre J., Folton N., Mathys N., Richard D. (2005): The hydrological impact of the mediterranean forest: a review of French research. Journal of Hydrology 301, 235–249

Croley II T. (1980): Gamma Synthetic Hydrographs. Journal of Hydrology, 47, 41-52.

Croley II T., Holly C.H. (1985):Resolving Thiessen Polygons. Journal of Hydrology, 76, 363-379.

Felleisen M., Findler R.B., Flatt M., Krishnamurthi S. (2001): How to Design Programs. An Introduction to Computing and Programming. The MIT Press. Cambridge.

Fischer, C. (2007): Hydrological Modeling of the Water Resources in the Nahal Harod, Israel. Diplomarbeit am Institut für Hydrologie der Albert-Ludwigs-Universität Freiburg i. Br.

Fread D.L. (1992): Chapter 10: Flow Routing. Handbook of Hydrology. Editor D. R. Maidment. McGraw Hill, New York.

Gaßmann M. (2007): Measuring and Modelling Erosion and Suspended Sediment. Diplomarbeit am Institut für Hydrologie der Albert-Ludwigs-Universität Freiburg i. Br.

Globevnik L., Sovinc A., Fazarinc R. (1998): Land degradation and environmental changes in a slovenian submediterranean area (Dragonja River-Catchment). Geoökodynamik, 19, 281-292.

Google Inc. (2007): Google Earth. Programmversion vom 12.9.2007, http://earth.google.com/intl/de/products.html.

Gunkel, A. (unveröffentlicht): Kopplung der Modelle ZIN und TRAIN (Arbeitstitel). Dissertation zur Erlangung des Doktorgrades der Geowissenschaftlichen Fakultät der Albert-Ludwigs-Universität Freiburg i. Br.

Helmke H., Höppner F., Iserhagen R. (2007): Einführung in die Software-Entwicklung. Hanser, München.

Horton R.E. (1933): The role of infiltration in the hydrological cycle. American Geophysical Union Transactions 14: 446-460.

Keestra S.D. (2007): Impact of natural reforestation on floodplain sedimentation in the Dragonja basin, SW Slovenia. Earth Surface Processes and Landforms, 32, 49–65.

Keesstra S.D., van Huissteden J., Vandenberghe J., Van Dam O., de Gier J., Pleizier I.D.(2005): Evolution of the morphology of the river Dragonja (SW Slovenia) due to land-use changes. Geomorphology. Volume 69, Issues 1-4, July 2005, Pages 191-207

Lange J. (1999): A non-calibrated rainfall-runoff model for large arid catchments, Nahal ZIN, Israel. Freiburg Schriften zur Hydrologie, Band 9. Institut für Hydrologie, Albert-Ludwigs-Universität Freiburg i. Br.

Lange J., Leibundgut C., Schick A.-C. (2000): The importance of Single Events in Arid Zone Rainfall-Runoff Modelling. Phys. Chem. Earth, Vol. 25, No 7-8, pp 673-677.

Leistert H. (2005): Modelling transmission losses, Applications in the Wadi Kuiseb and the Nahal ZIN. Diplomarbeit am Institut für Hydrologie der Albert-Ludwigs-Universität Freiburg i. Br.

Maniak U. (2005): Hydrologie und Wasserwirtschaft. Springer, Berlin.

Mardešić P., Dugački Z. (1962): Geografski Atlas Jugoslavije. Znanje, Zagreb.

Menzel L. (1999): Flächenhafte Modellierung der Evapotranspiration mit TRAIN. Potsdam-Instiut für Klimafolgenforschung (PIK), Potsdam, PIK Report No 54.

Nadarajah S. (2007): Probability Models for Unit-Hydrograph Derivation. Journal of Hydrology 344, 185 - 189.

Nash J., Sutcliffe J. (1970). River Flow Forecasting Through Conceptual Models. Part I - A Discussion of Principles. Journal of Hydrology, 10, 282-290.

Rao A.R., Hamed K.H. (2000): Flood Frequency Analysis. CRC Press LLC, Boca Raton.

Raws W.J., Ahuja, L.R., Brakensiek D.L., Shirmohammadi A. (1993). Handbook of Hydrology, chap. 5, McGraw Hill.

Saxton, K.E. (2007.): SPAW Soil-Plant-Atmosphere–Water Field & Pond Hydrology. (http://hydrolab.arsusda.gov/SPAW/Index.htm, 05.2007)

Schütz T. (2006): Prozessbasierte Niederschlag-Abflussmodellierung in einem mediterranen Kleineinzugsgebiet. Diplomarbeit am Institut für Hydrologie der Albert-Ludwigs-Universität Freiburg i. Br.

Shepard D. (1968): A two-dimensional interpolation function for irregularly-spaced data. Proceedings of the ACM national conference, p. 517-524. ACM Press.

Sherman L.K. (1932): Streamflow from rainfall by the unit hydrograph method. Engeneering News Record 108: 501-505.

Smith J.A. (1992): Chapter 3: Precipitation. Handbook of Hydrology. Editor D. R. Maidment. McGraw Hill, New York.

Stroustrup B. (1997): The C++ Programming Language. Addison Wesley Longman, Reading.

Sraj M., Brilly M., Globevnik L., Padeznik M., Mikos M. (2006): The Dragonja river experimental watershed. University of Ljubljana, Faculty of Civil and Geodetic Engineering, Slovenia. Geophysical Research Abstracts, Vol. 8, 02653.

Szél S., Csaba G. (2000): On the negative weighting factors in the Muskingum-Cunge scheme. Journal of Hydraulic Research, Vol. 38 No 4.

Szilagyi J. (1992): Why can the weighting parameter of the Muskingurn channel routing method be negative? Journal of Hydrology, 138, p. 145-151.

Thormählen, A.-C. (2003): Hydrological modelling in a small hyperarid catchment Nahal Yael, Israel – runoff generation an transmission losses. Diplomarbeit am Institut für Hydrologie der Albert-Ludwigs-Universität Freiburg i. Br.

Thornthwaite C.W. (1948): An Approach Toward a Rational Classification of Climate. Geographical Review, 38, 55 - 94.

Todini A. (2007): A mass conservative and water storage consistent variable parameter Muskingum-Cunge approach. Hydrology and Earth System Sciences.11, 1549-1592.

van der Tool C. (2006): Climatic constraints on carbon assimilation and transpiration of sub-Mediterranean forests. Doctoral thesis, Vrije Universiteit Amsterdam, The Netherlands.

Wagner A. (2002): Anwendung eines nicht-kalibrierten Niederschlag-Abfluss-Modells in den hydrologischen Versuchsgebieten Ostkaiserstuhl. Diplomarbeit am Institut für Hydrologie der Albert-Ludwigs-Universität Freiburg i. Br.

10.1 verwendete Software:

Microsoft Visual Studio 2003.

Microsoft Visual C++ Express, Versionen 2005 und 2008. Erhältlich unter http://www.microsoft.com/germany/express/default.aspx

Silverfrost FTN95PE, Version 5.10. Salford Software (Juni 2007). Erhältlich unter http://www.silverfrost.com/11/ftn95/overview.asp

SPAW, Version 6.02.74. Soil-Plant-Atmosphere-Water Field & Pond Hydrology. Saxton (Mai 2007). Erhältlich unter http://hydrolab.arsusda.gov/SPAW/Index.htm

Subversion, Versionen 1.4.5 und 1.4.6. Subversion open-source-community (August 2007, Dezember 2007). Erhältlich unter http://subversion.tigris.org/ oder als Bestandteil der Tortoise-Software.

Tortoise Client for SVN, Versionen 1.4.5 bis 1.4.8. Küng, S., Onken, L. (August 2007 - Februar 2008). Erhältlich unter http://tortoisesvn.net/node?page=1 .

Visual Leak Detector, Version 1.9.f (Beta). Moulding, D. (2006). Erhältlich unter http://dmoulding.googlepages.com/vld.

VSE debug, Version 1.0c, Mitchell, M. (April 2004). Erhältlich unter http://vsedebug.sourceforge.net/

WinMerge, Version 2.6.8. WinMerge open-source-community (Juni 2007). Erhältlich unter http://winmerge.org/.

Abkürzungen und Symbole

Abfluss	m^3/s	Q
Abfluss, gemessen	m^3/s	Q_{obs}
Abfluss, simuliert	m^3/s	Q_{sim}
Abflussdifferenz	m^3/s	Q_{diff}
American Standard Code for Information Interchange		ASCII
Bodenwassergehalt, aktuell		θ_{act}
Bodenwassergehalt bei Feldkapazität		θ_{FK}
Bodenwassergehalt beim permanenten Welkepunkt		θ_{PWP}
Durchlässigkeitsbeiwert	m/s	K_f
Feldkapazität	%	FK
Globalstrahlung	W/m^2	G
Gigahertz	1/s	Ghz
Höhe	m	h
Inverse-Distance-Weighting		IDW
Megahertz	1/s	Mhz
Muskingum-Cunge		MC
Nash-Sutcliffe-Koeffizient		R_{eff}
Niederschlag	mm	P
Niederschlagsintensität	mm/h	I_P
Niederschlagsintensität, maximale	mm/h	I_{P_max}
nutzbare Feldkapazität		nFK
Permanenter Welkepunkt	%	PWP
Speicherinhalt		S
Subversion-Software		SVN
Unit Hydrograph		UH
Verdunstung, aktuelle	mm	ETA
Visual Studio (© Microsoft Corp.)		VS

Anhang A – Abbildungen

Abb. 10.2 Brücke über die Dragonja nahe dem Pegel (Foto: Jens Lange)

Abb. 10.1 trockenes Flussbett im Einzugsgebiet (Foto: Jens Lange)

Abb. 10.3 Flussbett im Dragonja-Eizugsgebiet (Foto: Jens Lange)

Abb. 10.4 Flyschwand im Dragonja-Einzugsgebiet (Foto: Jens Lange)

Anhang B – Tabellen

Tabelle 11.2 Bodenparameter im ZIN-Modell

Nr	infCap (mm/h)	initLossCap (mm)	depth (m)	p_{eff} (%)	PWP	K_f (cm/h)	λ	FC (%)	Name	Anteil (%)
1	100	2	0,1	0,43	0,14	2	0,4	0	Pasture	8,7
2	200	2	0,5	0,477	0,14	8	0,4	0	Abandoned	18
3	200	2	0,8	0,477	0,14	3	0,4	0	Arable	12
4	200	2	1	0,477	0,14	3	0,4	0	young forest	8,1
5	200	3	1	0,477	0,14	5	0,4	0	forest	47
6	50	0,3	0,05	0,477	0,14	3	0,4	0	erosion	0,53
7	200	2	3	0,477	0,14	3	0,4	0	valleys	6,6

Tabelle 11.3 Gerinnenetz-Datei (Teil 1/4, Schlüssel „RiverNet” in der Steuerungsdatei)

seg	last	next	trib1	trib2	slope	length	width	type	show
1	0	3	0	0	0,0990	99	1	2	0
2	0	3	0	0	0.0990	99	1	2	0
3	1	4	2	0	0,0226	396	1,5	2	0
4	3	5	0	0	0,0371	381	2	2	0
5	4	7	0	0	0,0263	398	2,5	2	0
6	0	7	0	0	0,0990	99	1	2	0
7	5	8	6	0	0,0180	320	3	2	0
8	7	10	0	0	0,0146	313	3	2	0
9	0	10	0	0	0,0990	99	1,5	2	0
10	8	11	9	0	0,0177	283	3,5	2	0
11	10	20	0	0	0,0163	307	3,5	2	0
12	0	14	0	0	0,0990	99	1	2	0
13	0	14	0	0	0,0990	99	1	2	0
14	12	15	13	0	0,0351	333	1,5	2	0
15	14	16	0	0	0,0400	326	2	2	0
16	15	17	0	0	0,0357	320	2,5	2	0
17	16	18	0	0	0,0333	308	3	2	0
18	17	19	0	0	0,0318	332	3	2	0
19	18	20	0	0	0,0292	320	3	2	0
20	11	22	19	0	0,0139	480	3	2	0
21	0	22	0	0	0,0990	99	1	2	0
22	20	23	21	0	0,0122	313	3,5	2	0
23	22	25	0	0	0,0148	334	4	2	0
24	0	25	0	0	0,0990	99	1	2	0
25	23	26	24	0	0,0147	320	5	2	0
26	25	39	0	0	0,0146	346	6	2	0
27	0	29	0	0	0,0990	99	1	2	0
28	0	29	0	0	0,0990	99	1	2	0
29	27	30	28	0	0,0282	283	1,5	2	0
30	29	31	0	0	0,0402	323	2	2	0
31	30	32	0	0	0,0185	317	2,5	2	0
32	31	33	0	0	0,0191	308	2,5	2	0

Tabelle 11.4 Gerinnenetz-Datei (Teil 2/4)

seg	last	next	trib1	trib2	slope	length	width	type	show
33	32	34	0	0	0,0298	308	3	2	0
34	33	36	0	0	0,0278	351	3	2	0
35	0	36	0	0	0,099	99	3	2	0
36	34	37	35	0	0,0164	281	3	2	0
37	36	38	0	0	0,02	268	3	2	0
38	37	39	0	0	0,018	290	3	2	0
39	26	40	38	0	0,0137	337	6	2	0
40	39	42	0	0	0,0026	358	6,5	2	0
41	0	42	0	0	0,099	99	1	2	0
42	40	43	41	0	0,0266	244	6,5	2	0
43	42	45	0	0	0,0128	301	7	2	0
44	0	45	0	0	0,099	99	1	2	0
45	43	46	44	0	0,0135	332	7	2	0
46	45	48	0	0	0,0132	393	7	2	0
47	0	48	0	0	0,099	99	1	2	0
48	46	49	47	0	0,0154	277	7,5	2	0
49	48	50	0	0	0,013	301	7,5	2	0
50	49	52	0	0	0,0103	307	7,5	2	0
51	0	52	0	0	0,099	99	1	2	0
52	50	53	51	0	0,0106	253	7,5	2	0
53	52	55	0	0	0,0133	266	7,5	2	0
54	0	55	0	0	0,099	99	1	2	0
55	53	56	54	0	0,0088	298	8	2	0
56	55	57	0	0	0,0154	332	8	2	0
57	56	58	0	0	0,0119	335	8,5	2	0
58	57	60	0	0	0,0155	320	8,5	2	0
59	0	60	0	0	0,099	99	1	2	0
60	58	62	59	0	0,0139	280	9	2	0
61	0	62	0	0	0,099	99	1	2	0
62	60	63	61	0	0,0088	336	9	2	0
63	62	64	0	0	0,0224	362	9,5	2	0
64	63	65	0	0	0,0043	353	9,5	2	1

Tabelle 11.5 Gerinnenetz-Datei (Teil 3/4)

seg	last	next	trib1	trib2	slope	length	width	type	show
65	64	66	0	0	0,01	395	10	2	0
66	65	109	0	0	0,014	375	10	2	0
67	0	69	0	0	0,099	99	1	2	0
68	0	69	0	0	0,099	99	1	2	0
69	67	70	68	0	0,027	346	1,5	2	0
70	69	71	0	0	0,0281	355	1,5	2	0
71	70	72	0	0	0,0193	346	1,5	2	0
72	71	73	0	0	0,025	352	1,5	2	0
73	72	74	0	0	0,0136	352	1,5	2	0
74	73	75	0	0	0,017	295	1,5	2	0
75	74	76	0	0	0,0169	262	1,5	2	0
76	75	77	0	0	0,0171	336	1,5	2	0
77	76	78	0	0	0,0146	287	1,5	2	0
78	77	79	0	0	0,0136	290	1,5	2	0
79	78	80	0	0	0,0189	296	2	2	0
80	79	82	0	0	0,0187	298	2	2	0
81	0	82	0	0	0,099	99	1	2	0
82	80	83	81	0	0,0161	285	2	2	0
83	82	84	0	0	0,0161	263	2	2	0
84	83	85	0	0	0,0137	287	2	2	0
85	84	86	0	0	0,015	271	2	2	0
86	85	87	0	0	0,0129	274	2	2	0
87	86	88	0	0	0,0164	274	2	2	0
88	87	89	0	0	0,0249	268	2	2	0
89	88	90	0	0	0,0146	274	2	2	0
90	89	91	0	0	0,003	283	2,5	2	0
91	90	92	0	0	0,0084	277	2,5	2	0
92	91	93	0	0	0,0062	275	2,5	2	0
93	92	94	0	0	0,0173	287	2,5	2	0
94	93	95	0	0	0,0183	278	2,5	2	0
95	94	96	0	0	0,0203	272	2,5	2	0
96	95	97	0	0	0,0155	282	2,5	2	0

Tabelle 11.6 Gerinnenetz-Datei (Teil 4/4)

seg	last	next	trib1	trib2	slope	lengt	widt	type	show
97	96	99	0	0	0,0174	297	2,5	2	0
98	0	99	0	0	0,099	99	1	2	0
99	97	100	98	0	0,0051	253	2,5	2	0
100	99	101	0	0	0,0058	270	2,5	2	0
101	100	103	0	0	0,0272	256	2,5	2	0
102	0	103	0	0	0,099	99	1	2	0
103	101	104	102	0	0,0067	317	3	2	0
104	103	105	0	0	0,0232	322	3	2	0
105	104	107	0	0	0,0013	315	3	2	0
106	0	107	0	0	0,099	99	1	2	0
107	105	108	106	0	0,0143	447	3	2	1
108	107	109	0	0	0,0182	510	3,5	2	0
109	66	0	108	0	0,0109	402	11	2	1

Anhang C – Quellcode

C.1 Klasse Controller

(Code neu erstellt)

Inhalt der Datei Controller.cpp:

```
#include "stdafx.h"
#include <sstream>

//**** class by Uwe Hagenlocher****//
/* This class is for the communication with the ascii-interface-file. For using it
place in the ctr-file a keyword followed by the item the model needs. Use the
public Methods getPath(string), getBool(string) etc. to get the item you want.
There is only one instance of this class created at the very start of the Program.
The !reference! to this one object is handed over to all other classes on construction
of an object. So do not try to modify that object once its there!*/

Controller::Controller(string inFile){
      /* Constructor for a Controller-Object.
      The path and name of the ctr-file is the parameter thats being used
      to write the relevant part of the file to the istringstream ctrReader*/
      //    int readpos = 0;
      char stop;
      this->ctrFile = inFile;
      ifstream ctrStream;
      string ctrIn;                     //dummy for the reading of single tokens from
the ctr-file
      ctrFile = this->checkPath(ctrFile);      //check the file is there
      ctrStream.open(ctrFile.c_str());  // open file (needs a c_string)
      if (!ctrStream)                         //file not found in spite of checking
before?
            throw ReportException(this->ctrFile, 1);
      ctrStream >> ctrIn;               // read one token...
      if (ctrIn.compare("ctr"))   // file does not start with "ctr"?
            throw ReportException(this->ctrFile, 2);
      //overread everything up to the word "startcoding":
      while (ctrStream >> ctrIn && ctrIn.compare("startcoding"));
      ostringstream inputString(ios::in);
      while (ctrStream){
            stop = ctrStream.peek();          // look at, but do not extract the next
char
            while (stop == ' ' || stop == '\n' || stop == '\t'){   // look for the
next "word"
              stop = ctrStream.get();
              stop = ctrStream.peek();
            }
            if (stop == '%'){     // comment-line, continue with next line
              getline(ctrStream, ctrIn);
              continue;
            }
            if (ctrStream.eof())
              break;
            ctrStream >> ctrIn;                        // read from the fstream to the
string...
            inputString << ctrIn << " ";       // ... and from the string to the
stringstream
      }
      ctrStream.close();
      this->ctrreader.str(inputString.str()); // initialize ctrreader with the content
of inputString
}

Controller::~Controller(){
}

// private: extracts a path-, int-, float-, double- or bool-
```

```
// element from the ctr-file as a string:
string Controller::getElement(string elemCode){
      string ctrIn;
      this->ctrreader.clear();                  // remove error flags if any
      this->ctrreader.seekg(ios::beg);   // set the reading position to the beginning
of the stream
      /* while the end of the input-Stream is not reached and the
      requested codeword is not found, the next token is read into ctrIn: */
      while (ctrreader >> ctrIn && ctrIn.compare(elemCode));
      if (!ctrreader)             // end of inputStream is being reached without fin-
ding the codeword?
            throw ReportException(elemCode, 5);
      ctrreader >> ctrIn;  // this actually reads the Element as the token following
the codeword
      return ctrIn;
}

string Controller::getPath(string pathCode){
      // public: returns a checked pathname from the ctr-file
      string pathName;
      pathName = this->getElement("Proj_Fold") + this->getElement(pathCode);
      pathName = this->checkPath(pathName);
      return pathName;
}

string Controller::checkPath(string name){
      // (public) checks if "name" is valid and returns it if so
      // if not it returns a new path:
      if (!this->pathExists(name, 1))
            return this->getNewFilename(name);
      return name;
}

void Controller::checkWrite(string name){
      string reader;
      // until writing to name is possible or the program is stopped by entering "q":
      while (!this->pathExists(name, 2)){
            cout << "\a\nunable to write file: " << name << endl;
            cout << "possible reasons:\n"
              "- the file is in use by another application\n"
              "- you do not have the permission to write to the specified location\n"
              "- the folder in that you try to write does not exist\n"
              "press enter to retry or q to terminate TRAIN-ZIN:\n";
            getline(cin, reader);
            if (reader == "q")
              throw ReportException(name, 1);
      }
}

bool Controller::pathExists(string testThis, int readWrite){
      /* if "testThis" is a directory: check if this path exists
      if "testThis" is a path to a file:
      if readWrite equals 1 try to read the file,
      if readWrite equals 2 try to write in the file*/
      ifstream testIn;
      ofstream testOut;
      string tmpFile = "__deleteThisFile.xyz";
      // if testThis is a file, i.e. the last char of it is not "\" or "/"
      if (testThis.substr(testThis.size()-1).compare("\\") &&
            testThis.substr(testThis.size()-1).compare("/")){      // file, not direc-
tory
              if (readWrite == 1) // file is needed for reading
                    testIn.open(testThis.c_str());
              else{
                    testOut.open(testThis.c_str(), ios::app);
              }
            }
      else  // else testThis is a directory
            testOut.open((testThis + tmpFile).c_str());     // try to create a file in
that directory
      if (testIn.is_open() || testOut.is_open()){
            testIn.close();
```

```
            testOut.close();
            remove((testThis + tmpFile).c_str());     // delete the testfile
            return true;
      }
      else
            return false;
}

string Controller::getNewFilename(string oldName){
      // if a file is not found, ask the user for the right one and return that
      // or return an Exception if that is not posible (on entering "q")
      string newName;
      // until a valid path/file is found or the program is stopped by entering "q":
      do{
            cout << "\apath or file not found: " << oldName << endl;
            cout << "enter new path- or filename or q to terminate TRAIN-ZIN:\n";
            cin >> newName;
            if (newName == "q")
              throw ReportException(oldName, 1);
            oldName = newName;
      }while (!this->pathExists(newName, 1));
      return newName;
}

string Controller::getString(string whichString){
      return this->getElem<string>(whichString);
}

bool Controller::getBool(string whichBool){
      return this->getElem<bool>(whichBool);
}

int Controller::getInt(string whichInt){
      return this->getElem<int>(whichInt);
}

float Controller::getFloat(string whichFloat){
      return this->getElem<float>(whichFloat);
}

int Controller::getDateItem(const string& dateStr, const unsigned item){
      if (dateStr.size() == 0)
            return -1;
      int result;
      string dum;
      char num;
      istringstream conv;
      istringstream reader(dateStr);
      for (int i = 0; item - i > 0; i++){
            dum = "";
            while(reader >> num && num != '.'){
              dum += num;
            }
      }
      conv.str(dum);
      conv >> result;
      if (!conv)
            return -1;
      if (item == 3){ // year, make sure its 4-digit:
            if (result < 1000){          // year is 4-digit
              if (result > 30)    // assuming a date 30 + indicates 20th century...
                    result += 1900;
              else                // whereas a date <= 30 indicates 21st century (sor-
ry, 2031-guys!)
                    result += 2000;
            }
      }
      return result;
}

unsigned Controller::getJulDay(const string &dateStr){
      string dateCopy = dateStr;
```

```
      int dayOmonth;
      int month;
      int year;
      unsigned julDay = 0;
      unsigned febD;
      dayOmonth      = this->getDay(dateStr);
      month   = this->getMonth(dateStr);
      year    = this->getYear(dateStr);
      if (year % 4)
            febD = 28;
      else
            febD = 29;
      unsigned daysInMonth[] = {0, 31, febD, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      if (dayOmonth > daysInMonth[month] ||
            dayOmonth < 1 ||
            month < 1 ||
            month > 12 ||
            year < 1900 ||
            year > 2050)
            throw ReportException("bad date: " + dateCopy, 4);
      for (unsigned i = 1; i < month; i++)
            julDay += daysInMonth[i];
      julDay += dayOmonth;
      return julDay;
}

int Controller::getDay(const string &dateStr){
      return this->getDateItem(dateStr, 1);
}

int Controller::getMonth(const string &dateStr){
      return this->getDateItem(dateStr, 2);
}

int Controller::getYear(const string &dateStr){
      return this->getDateItem(dateStr, 3);
}

string Controller::getDateString(int doy, int year){
      int febD;
      int month = 1;
      int daysInMonth[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      ostringstream conv(ios::in);
      do{
            if (year % 4)
              febD = 28;
            else
              febD = 29;
            daysInMonth[2] = febD;
            while (doy - daysInMonth[month] > 0){
              doy -= daysInMonth[month];
              month++;
            }
      }
      // while serves as: if month > 12 increase year, set month to 1 and repeat the
calculation:
      while(month > 12 && year++ && (month = 1));
     conv << doy << "." << month << "." << year;
      return conv.str();
}

string Controller::getHHmmSS(double time){
      // converts a time in minutes to a string in hh:mm:ss format
      ostringstream conv(ios::in);
      int hh, mm, ss;
      while (time >= 1440)
            time -= 1440;  // time is more than one day
      hh = int (time / 60);
      time -= hh * 60;
      mm = int(time);
      time -= mm;
      time *= 60;    // convert time min to sec
```

```
      ss = int (time + 0.5);      // round the seconds
      conv << hh << ":" << mm << ":" << ss;
      return conv.str();
}

int Controller::dateCompare(const string& dateStr1,const string& dateStr2){
      /* compares 2 dates in format d.m.yy or d.m.yyyy
      and returns -1 if 1 < 2, +1 if 1 > 2 and 0 for 1 == 2 */
      int dat1, dat2;
      for (unsigned i = 3; i > 0; i--){  // compare...
            dat1 = this->getDateItem(dateStr1, i);
            dat2 = this->getDateItem(dateStr2, i);
            if (dat1 < 0 || dat2 < 0)
              throw ReportException("wrong date Format", 4);
            if (dat1 > dat2)
              return 1;
            if (dat1 < dat2)
              return -1;
      }
      return 0;      //equal
}
```

Inhalt der Datei Controller.h (Auszug):

(Quellcode neu erstellt)

```
private:
      template<typename T> T getElem(string key){
            string ctrIn;
            this->ctrreader.clear();           // remove error flags if any
            this->ctrreader.seekg(ios::beg);// set the reading position to the begin-
ning of the stream
            /* while the end of the input-Stream is not reached and the
            requested codeword is not found, the next token is read into ctrIn: */
            while (ctrreader >> ctrIn && ctrIn != key);
            if (!ctrreader)             // end of inputStream is being reached without
finding the codeword?
              throw ReportException(key, 5);
            ctrreader >> ctrIn;   // this actually reads the Element as the token
following the codeword
            T res;
            istringstream conv(this->getElement(key));
            if (!(conv >> res)){
              conv.clear();
              cout << "bad content: \"" << conv.str()
                    << "\" for Controller-Element \"" << key << "\"!\n";
              cout << "enter new value or q to terminate TRAIN-ZIN: ";
              cin >> res;
              if (!cin)
                    throw ReportException(key, 5);
            }
            return res;
      }
```

C.2 Klasse Grid

Methode Grid::read():

(bestehender Code wurde stark überarbeitet, das Einlesen von `float-` und `double-`Grids ist analog dem Einlesen von `integer-`Grids und wurde deswegen weggelassen)

```
void Grid::read(string filename){
      // method for opening a file and placing the data into a matrix
      ifstream in; //used for opening the file
```

```
	filename = this->gridCont->checkPath(filename);
	in.open(filename.c_str()); //opens the file
	if (!in)
		throw ReportException(filename, 1);
	cout << "reading " << filename << " ... ";
	string strIn;  //string for input of a line U: dummy only
	double doubIn;		//dummy
	//U: constructing the Matrix:
	if (varType != "int" && varType != "float" && varType != "double")
		throw ReportException(filename, 2);
	/* if the file contains x- and ySize use them to verify the file contains the
right number of
	rows/cols, then eat rest of header if present.
	This works for ArcGis Headers where the first token of each line is no double
(or float or int),
	no matter what (or if anything) follows in that line, so the first line starting
with
	a number is the starting point for reading. In the for- loop: As the first Ele-
ment has
	been extracted allready, adding to the matrix has to happen _before_ reading and
stop
	right before reaching [xMax, yMax]!*/
	in >> doubIn;
	while (!in){		// !input because line is a header and no double
		in.clear();		// remove error flag
		in >> strIn;
		if (strIn == "ncols"){
		  in >> doubIn;
		  if (int(doubIn + 0.5) != this->xSize)
			throw ReportException(filename, 2);
		  in >> strIn;
		}
		if (strIn == "nrows"){
		  in >> doubIn;
		  if (int(doubIn + 0.5) != this->ySize)
			throw ReportException(filename, 2);
		}
		getline(in, strIn);	// eat this header-line
		in >> doubIn;		// try to read beginning of next line
	}
	//Integers:
	if (varType == "int"){
		int dummy = int (doubIn - 0.5);
		for (int y = ySize - 1; y >= 0; y--){
		  for (unsigned x = 0; x < xSize; x++){
			/* the first token has allready been read, so stop one token short
			of the end and 'put' before 'read':*/
			if (!(x + 1 == xSize && y == 0)){
				this->dataInt[y * this->xSize + x] = dummy;
				in >> dummy;
			}
		  }
		}
		this->dataInt[this->xSize - 1] = dummy;
	}
	//Floats:
		...
	//Doubles:
		...
	if (!in)	// having tried to read past eof?
		throw ReportException(filename, 2);
	in >> doubIn;
	if (in)		// file contains more data?
		throw ReportException(filename, 2);
	in.close();
	cout << "\tdone!" << endl;
}
```

Methode Grid::setHeader()

(Code wurde neu erstellt)

```
void Grid::setHeader(){
	string aux12b;		/*this variable was named in grateful memory of all the
people who spent
				precious time devising meaningful variable names in this
program*/
	double  val;
	bool fail = false;	// error in reading the header file
	aux12b = this->gridCont->getPath("header");
	ifstream in(aux12b.c_str());
	if (!in)
		throw ReportException(aux12b, 1);
	// check if the keywords are correct:
	in >> aux12b;
	fail = (aux12b != "ncols"		|| fail);	// sets error (or retains prior
failure)
	getline(in, aux12b);
	in >> aux12b;
	fail = (aux12b != "nrows"		|| fail);
	getline(in, aux12b);
	in >> aux12b;
	fail = (aux12b != "xllcorner"	|| fail);
	getline(in, aux12b);
	in >> aux12b;
	fail = (aux12b != "yllcorner"	|| fail);
	getline(in, aux12b);
	in >> aux12b;
	fail = (aux12b != "cellsize"	|| fail);
	getline(in, aux12b);
	in >> aux12b;
	fail = (aux12b != "NODATA_value" || fail);
	if (fail)
		throw ReportException(aux12b, 2);
	// all keywords ok, start reading:
	in.seekg(ios::beg);
	in >> aux12b;		// ignore keyword
	in >> val;
	this->xSize		= int(val);
	in >> aux12b;
	in >> val;
	this->ySize		= int(val);
	in >> aux12b;
	in >> val;
	this->xllcorner = val;
	in >> aux12b;
	in >> val;
	this->yllcorner = val;
	in >> aux12b;
	in >> val;
	this->cellsize	= val;
	in >> aux12b;
	in >> val;
	this->noData   = int(val - 0.5);
	in.close();
}
```

Methode writeGrid(string):

(Methode auf `string` umgestellt, um Ausgabe des headers erweitert, Sicherheitsabfragen erstellt)

```
void Grid::writeGrid(string filename){
	/* write a Grid to the specified file */
	filename += ".asc";
	if (!(this->varType == "int" || this->varType == "float" || this->varType ==
"double"))
```

```
            throw ReportException(filename, 2);
      this->gridCont->checkWrite(filename);
      ofstream out(filename.c_str());
      if (!out)
            throw ReportException(filename, 1);
      out.precision(15);   // write numbers with (i) digits (before + after:
123.45678)
      cout << "writing file " << filename << " ... ";
      out << "ncols\t"            << this->xSize            << endl;
      out << "nrows\t"            << this->ySize            << endl;
      out << "xllcorner\t" << this->xllcorner  << endl;
      out << "yllcorner\t" << this->yllcorner  << endl;
      out << "cellsize\t"         << this->cellsize   << endl;
      out << "NODATA_value\t" << this->noData         << endl;
      out.precision(8);
      if (this->varType == "int"){
            for (int y = ySize - 1; y >= 0; y--){
              for (unsigned x = 0; x < xSize; x++)
                    out << this->dataInt[y * this->xSize + x] << " ";
              out << endl;
            }
      }
      if (this->varType == "float"){
            for (int y = ySize - 1; y >= 0; y--){
              for (unsigned x = 0; x < xSize; x++)
                    out << this->dataFloat[y * this->xSize + x] << " ";
              out << endl;
            }
      }
      if (this->varType == "double"){
            for (int y = ySize - 1; y >= 0; y--){
              for (unsigned x = 0; x < xSize; x++)
                    out << this->dataDouble[y * this->xSize + x] << " ";
              out << endl;
            }
      }
      out.close();
      cout << "\tdone!" << endl;
}
```

C.3 Klasse QConc

(Klasse vollständig überarbeitet, nur Reste von altem Code und Struktur enthalten)

```
#include "stdafx.h" //precompiled header
#include "QConc.h"

/* runoff concentration*/
QConc::QConc(unsigned length, string evName, Controller* qCont){
      this->qConcCont      = qCont;
      this->polynum  = length;                  //number of runoff generation polygons
(subbasins)
      this->eventName      = evName;
      this->zinStep  = this->qConcCont->getFloat("ZINStep");
      this->zinSteps       = unsigned (1440.0 / zinStep + 0.5); // number of zinSteps
a day (rounded)
      this->chanseg  = unsigned (this->qConcCont->getInt("ChanNumber")); //number of
channel segments
      this->segno          = new int[chanseg + 1];    //array for storing the channel
segment numbers
      this->lpoly          = new int[chanseg + 1];    //array: polygon number at the
right side of a segment
      this->latq.resize(chanseg + 1);                 // initializing the size of
outer vector, thats segments!
      this->latql          = new double*[chanseg + 1];      //runoff from the polygon
for time and channel segement
      this->routeStep = this->qConcCont->getFloat("RouteStep");
      this->routeSteps= unsigned(1440 / this->routeStep + 0.5); //+0.5 for correct
rounding to integer
      if (this->qConcCont->getInt("RainMethod") == 1)
```

```
            this->isRadar = true;
      else isRadar = false;
      /*sets the zero values:*/
      for (unsigned j = 0; j <= chanseg; j++){
            latql[j]       = new double[this->zinSteps + 1];
            for (unsigned t = 0; t <= this->zinSteps; t++)
              latql[j][t]  = 0;
      }
      this->methd    = this->qConcCont->getInt("QConcMethod");
      this->smooth= this->qConcCont->getBool("flatten_conc");
      switch (methd){
            case 0:
              this->setupMeasured();
              break;
            case 1:
              this->setupFish();
              break;
            default:
              throw ReportException("QConcMethod not valid!", 4);
      }
}

QConc::~QConc(){
      delete [] segno;
      segno = 0;
      delete [] lpoly;
      lpoly = 0;
      for (unsigned j = 0; j <= chanseg; j++){
            if (latql){
              delete [] latql[j];
              latql[j] = 0;
            }
      }
      delete [] latql;
      latql = 0;
}

void QConc::setupMeasured(){
      // read the subcatchment-data:
      float  fDum;
      double  reader;
      double  sum = 0;
      string  strDum;
      ifstream seginput(this->qConcCont->getPath("Ezg_Seg").c_str());
      seginput >> strDum;
      this->curveRes = 1;
      getline(seginput, strDum);                  // overread header-line
      for (unsigned j = 1; j <= chanseg; j++)
            latq[j].resize(int((2 * 1440/this->routeStep) + 1.5), 0);
      for (unsigned i = 1; i <= chanseg; i++){
            seginput >> this->segno[i];           // number of the channel segment
            seginput >> this->lpoly[i];           // number of the polygon
            getline(seginput, strDum);
      }
      if (!seginput)
            throw ReportException("EzgSeg", 2);
      seginput.close();
      this->distCurve.resize(2);                  // create a curve with index 1 (and one
with 0)
      ifstream in(this->qConcCont->getPath("runoffCrv").c_str());
      if (!in)
            throw ReportException("runoffCrv", 2);
      this->distCurve[1].push_back(0);   //create curve-element 0 so the curve starts
at 1
      getline(in, strDum);                        // overread header-line
      while (in){
            in >> fDum;                                       // number of timestep or
minutes, not needed
            in >> reader;
            if (!in)
              break;
            this->distCurve[1].push_back(float(reader));
```

```
            sum += reader;
      }
      in.close();
      if (abs(1 - sum) > 0.01){
            cout << "\nwarning: sum of measured Q-concentration-curve is " << sum <<
"\n"
              "while a value of 1 +- 0.01 was expected.\n"
              "enter q to terminate TRAIN-ZIN\n"
              "or any other key to proceed and auto-correct the f(x) values: ";
            getline(cin, strDum);
            if (strDum == "q")
              throw ReportException("runoffCrv", 2);
            if (sum == 0)
              throw ReportException("Sum of concentration curve-factors is 0!", 4);
      }
      for (unsigned distStep = 1; distStep < this->distCurve[1].size(); distStep++)
            this->distCurve[1][distStep] /= sum; // correct the values so the sum is 1
}     // end measured

void QConc::setupFish(){
      //setupt for an Extremal-value distribution Type I (or Gumbel or fisher-Tip-
pett):
      const double EulerMasch = 0.577215664901533;    // Euler-Mascheroni-constant
      const int TimesStdDev = 8;        /* factor to be multiplied with the standard
Deviation to controll
                                                      the length of the tailing*/
      const float PI = 3.14159265F;
      // zeitschritt: number of timesteps the runoff of the subcatchment is distribu-
ted to:
      unsigned zeitschritt;
      unsigned stepMax = 0;
      // read the subcatchment-data:
      double area_avr      = 0;         // average area  of the subcatchments
      double slope_avr = 0;             // average slope of the subcatchments
      double mean;                      // mean of fisher-tippett-distribution
      double stdDev;                          // standard Deviation of fisher-
tippett-distribution
      float slopeDep       = this-> qConcCont->getFloat("SlopeDep");
      float areaDep  = this-> qConcCont->getFloat("AreaDep");
      float fishAraw       = this->qConcCont->getFloat("fishT_a") / this->routeStep;
// convert from minutes to timesteps
      float fishBraw       = this->qConcCont->getFloat("fishT_b") / this->routeStep;
      float*  segslope     = new float[this->polynum + 1];
      float*  segarea            = new float[this->polynum + 1];
      double  fish_f;                         // value of                distribu-
tion function f(x)
      double  fish_F;                         // value of cumulated distribution
function F(x)
      double*       fish_a       = new double[this->polynum + 1];
      double*       fish_b       = new double[this->polynum + 1];
      ifstream seginput(this->qConcCont->getPath("Ezg_Seg").c_str());
      string strDum;
      getline(seginput, strDum);              // overread header-line
      for (unsigned i = 1; i <= this->chanseg; i++){
            seginput >> segno[i];                     // number of the channel segment
            seginput >> lpoly[i];                     // read the index of related
subbasin
            seginput >> segslope[lpoly[i]];   // slope of the basin
            seginput >>  segarea[lpoly[i]];   // area of the polygon in whatever
units
            area_avr       += segarea[lpoly[i]];
            slope_avr      += segslope[lpoly[i]];
      }
      if (!seginput)
            throw ReportException("EzgSeg", 2);
      seginput.close();
      area_avr       /= polynum;
      slope_avr      /= polynum;
      this->distCurve.resize(polynum + 1);
      for (unsigned seg = 1; seg <= chanseg; seg++)
            latq[seg].resize(int((2 * 1440/this->routeStep) + 1.5), 0);   // resize to
two days + (1 routingStep)
```

```
      for (unsigned pol = 1; pol <= polynum; pol++){
            fish_a[pol] = fishAraw / (1 + slopeDep / 100 *
              (segslope[pol] - slope_avr) / slope_avr);
            fish_b[pol] = fishBraw * (1 + areaDep / 100 *
              (segarea[pol] - area_avr) / area_avr);
            mean = fish_a[pol] + fish_b[pol] * EulerMasch;
            stdDev = PI * fish_b[pol] / sqrt(6.0);
            /* +1 because it takes at least one timestep to get the water to the chan-
nel: */
            zeitschritt = 1 + unsigned(mean + stdDev * TimesStdDev);
            stepMax = max(zeitschritt, stepMax);      // longest tailing of any poly-
gon, used for output
            // distCurve[pol][0] is created so the first .push_back in the loop acces-
ses ...[1]:
            this->distCurve[pol].resize(1);
            fish_F = 0;
            for (unsigned distStep = 1; distStep <= zeitschritt; distStep++){
              fish_f = this->fisherTippet(distStep, fish_a[pol], fish_b[pol]);
              this->distCurve[pol].push_back(fish_f);
              fish_F += fish_f;
            }
            for (unsigned distStep = 1; distStep <= zeitschritt; distStep++){
              this->distCurve[pol][distStep] /= fish_F;      // corect the values so
the sum is 1
            }
      }
      // output of the Fisher-Tippett-curves:
      string fishFile;
      fishFile = this->qConcCont->getPath("Output") + "FiTiCurves.txt";
      this->qConcCont->checkWrite(fishFile);
      cout << "writing " << fishFile << " ... ";
      ofstream fishout(fishFile.c_str());
      fishout << "average area:\t" << area_avr << "\naverage slope:\t" << slope_avr <<
endl;
      fishout << "polygon:\t";
      for (unsigned pol = 1; pol <= polynum; pol++){
            fishout << pol << "\t";
      }
      fishout << endl;
      fishout << "a:\t";
      for (unsigned pol = 1; pol <= polynum; pol++){
            fishout << fish_a[pol] << "\t";
      }
      fishout << endl;
      fishout << "b:\t";
      for (unsigned pol = 1; pol <= polynum; pol++){
            fishout << fish_b[pol] << "\t";
      }
      fishout << endl;
      fishout << "area:\t";
      for (unsigned pol = 1; pol <= polynum; pol++){
            fishout << segarea[pol] << "\t";
      }
      fishout << endl;
      fishout << "slope:\t";
      for (unsigned pol = 1; pol <= polynum; pol++){
            fishout << segslope[pol] << "\t";
      }
      fishout << endl;
      for (unsigned t = 1; t < stepMax; t++){
            fishout << t*this->routeStep/1440 << "\t";      //time as fraction of day
            for (unsigned pol = 1; pol <= polynum; pol++){
              if (this->distCurve[pol].size() <= t)
                    fishout << 0 << "\t";
              else
                    fishout << this->distCurve[pol][t] << "\t";
            }
            fishout << endl;
      }
      fishout.close();
      delete [] segslope;
      segslope = 0;
```

```
      delete [] segarea;
      segarea = 0;
      delete [] fish_a;
      fish_a = 0;
      delete [] fish_b;
      fish_b = 0;
      cout << "\tdone!" << endl;
      // end of output of the Fisher-Tippett-curves
}     // end fisher-Tippett-setup

void QConc::calcLatq(double** qsum, unsigned chanseg, string totalName, int** con-
versArray){
      /* calculates runoff per channel segment based on runoff per polygon
      transform generated runoff to the channel */
      int     cn;           //index of the curve that is used
      int     left;
      unsigned              eraseSteps;
      unsigned              vectorSize;
      int     eraseMe;
      //float       sum    = 0;
      //    float   qC     = 5e+4;
      long  routeTime;
      double  tmp;
      double** polyq;
      polyq = qsum; //pointer to Runoff::sumRunoff[][] (as long as that exists...)
      this->totalSum = 0;
      if (this->methd == 1 || this->curveRes == 1)
            eraseSteps = unsigned (1440.0 / this->routeStep + 0.5);
      else
            eraseSteps = this->zinSteps;
      for (unsigned j = 1; j <= chanseg; j++){
            // discard the previous day (timesteps [1] to eraseSteps):
            vectorSize = latq[j].size() - 1;
            eraseMe = min(eraseSteps, vectorSize);
            latq[j].erase(latq[j].begin() + 1, latq[j].begin() + eraseMe + 1);
            switch (this->methd){
                  case 0:             // measured, only one curve for all sub catch-
ments
                        cn = 1;
                        break;
                  default:      // distribution, one curve for every subcatchment
                        cn = lpoly[j];
            }
            // vectorSize: desired size of the vector, one day plus the length of the
Qconc-curve:
            vectorSize = unsigned (1440 / this->routeStep + 1.5);  // add one and
round to closest int
            vectorSize += this->distCurve[cn].size();
            // make sure the vector contains data for at least this day:
            latq[j].resize(vectorSize, 0);
      }
      cout << "runoff-concentration for ZIN-timestep    ";
      /*calculate the runoff in all channel segments (second loop) for each zin-times-
tep (first loop):*/
      for (unsigned t = 1; t <= this->zinSteps; t++){       // timesteps day
            routeTime = int ((t - 1) * zinStep / routeStep + 0.5);
            for (int z = 1; z <= t; z *= 10)
              cout << "\b";
            cout << t;
            for (unsigned segment = 1; segment <= chanseg; segment++){    // channel
segments
              left = lpoly[segment];     //place where the leftside polygon is stored
in the runoff array
              if (left == 0){
                    latq[segment][t] = 0; //else no runoff added to the channel seg-
ment
                    continue;     // no runoff, so go to next segment
              }
              latql[left][t] = polyq[t][left]; //gets the runoff for this polygon
(actual timestep)
              //sum += polyq[t][left];
              switch (this->methd){
```

```
                        case 0:             // measured, only one curve for all sub catch-
ments
                              cn = 1;
                              break;
                        default:      // distribution, one curve for every subcatchment
                              cn = left;
                  }
                  // check the size of the vector, extend if too small:
                  //if (latq[segment].size() <= routeTime + distCurve[cn].size() - 1)
                  //     latq[segment].resize(      routeTime + distCurve[cn].size(), 0);
                  vectorSize = this->distCurve[cn].size();
                  for (unsigned i = 1; i < vectorSize; i++){
                        // index: ZIN-time converted to routing-time + delay-time (all in
steps):
                        tmp = latql[left][t] * this->distCurve[cn][i];
                        //try{
                        latq[segment][routeTime + i] += tmp;
                        this->totalSum += tmp;
                        //}
                        //catch(...){
                        //     /* having tried to violate the vector bounds?
                        //     resize the vector and decrease i to try again: */
                        //     latq[segment].resize(routeTime + distCurve[cn].size(), 0);
                        //     i--;
                        //}
                  }
            }           // end segments segment
      }     // end ZIN-steps t
      cout << endl;
      //this->output(chanseg, totalName + "_rough", conversArray); // write to file
      if (smooth)
            this->flatten();
      this->output(chanseg, totalName, conversArray); // write to file
}     // end calcLatq

double QConc::getAverage(unsigned seg, int iniStep, int ratio){
      double average = 0;
      if (iniStep < 1)
            return -1;
      if (iniStep + ratio >= this->latq[seg].size())
            return -1;
      for (int i = iniStep; i < iniStep + ratio; i++){
            average += this->latq[seg][i];
      }
      return average / ratio;
}

double QConc::getSteepness(int ratio, double avLast, double avThis, double avNext){
      /* for smoothing the result, this gets the steepness of the line to be used as
      the mean of the steepness (from last to this) and (this to next zin-step)*/
      if (avThis < 0)
            return -1;
      if (avThis == 0)
            return 0;
      if (avLast < 0)
            return (avNext - avThis) / ratio;
      if (avNext < 0)
            return  (avThis - avLast) / ratio;
      return (      (avNext - avThis) / ratio + (avThis - avLast) / ratio) / 2;
}

void QConc::flatten(){
      double avLast;              // average runoff-value of last, this and next ZIN-
step
      double avThis;
      double avNext;
      double steepness;
      unsigned      stepsRatio;         // routeStep * stepsRatio == ZINStep
      stepsRatio    = int(this->zinStep / this->routeStep + 0.5);
      for (unsigned segment = 1; segment < latq.size(); segment++){
            for (unsigned roStep = 1; roStep + stepsRatio < this-
>latq[segment].size(); roStep += stepsRatio){
```

```
                avLast = this->getAverage(segment, roStep - stepsRatio, stepsRatio);
                avThis = this->getAverage(segment, roStep                       , steps-
Ratio);
                if (avThis < 0)
                        continue;
                avNext = this->getAverage(segment, roStep + stepsRatio, stepsRatio);
                steepness = this->getSteepness(stepsRatio, avLast, avThis, avNext);
                for (unsigned res = roStep; res < roStep + stepsRatio; res++){
                        latq[segment][res] = avThis + steepness * (res - roStep + 1 -
(stepsRatio + 1.0) / 2.0);
                        if (latq[segment][res] < 0){      // runoff negative?
                                if (steepness == 0)
                                        continue;                       // runoff is nega-
tive for other reasons
                                steepness *= 0.8;                 // reduce steepness
                                if (abs(steepness) < 0.000001)
                                        steepness = 0;
                                res = roStep - 1;                       // start over again
                        }
                  }
                }
        }
}

vector<vector<double> > & QConc::getLatqRef(){
        /* returns a _reference_ !not the vector itself!
        the reference is used in the routing */
        return this->latq;
}

double QConc::fisherTippet(int step, double fA, double fB){
        if (fB <= 0)
                throw ReportException("Fisher-Tippet-parameter b must not equal 0!", 4);
        double fx;      // f(x)
        double xMinAdivB;
        xMinAdivB = (step - fA) / fB;
        if (xMinAdivB < -4)  // yields a *very* small number for fx, can be too much for
double precision!
                return 0.0;
        fx = 1 / fB * exp(-xMinAdivB) * exp(-exp(-xMinAdivB));
        return fx;
}

void QConc::output(unsigned chanseg, string totalName, int** conversArray){
        /*write the output file (first column: segment number; second column: runoff per
timestep)
        conversArray is a pointer to Execution::RadarRecord*/
        string file;
        file = this->qConcCont->getPath("Outfold_lat") + "latro";
        file += totalName;
        file += ".txt";
        this->qConcCont->checkWrite(file);
        fstream roout(file.c_str(), ios::out);
        roout.seekg(0);
        roout.precision(8);
        cout << "writing file: " << file << " ... ";
        int conversion;
        /*let the line with segments begin in the second col as the first is used for
minutes:*/
        roout << "\t";
        for (unsigned i = 1; i <= chanseg; i++)
                roout << '\'' << segno[i] << "\t";
        roout << endl;
        conversion = int (this->zinStep / this->routeStep + 0.5);
        if (this->curveRes == 1)
                conversion = 1;
        int radStep = 0;
        float radTime = this->routeStep;
        for (unsigned t = 1; t <= routeSteps; t++){
                if (this->isRadar){
                  if (fabs(radTime - t * this->routeStep) < 0.001){
                        radStep++;
```

```
                        radTime += conversArray[radStep][1];
                        conversion = int (conversArray[radStep][1] / this->routeStep +
0.5);
                  }
                }
                roout << t * this->routeStep << "\t";
                for (unsigned i = 1; i <= chanseg; i++){
                  if (t >= latq[i].size()){
                        roout << 0;
                  }
                  else
                        roout << latq[i][t] / conversion;
                  roout << "\t";
                }
                roout << endl;
                if (!roout)
                  throw ReportException(file, 6);
        }
        roout.close();
        cout << "\tdone!" << endl;
}       // end output


int QConc::getChanseg(){
        return chanseg;
}

double QConc::getQCsum(){
        return this->totalSum;
}

double QConc::getQsum_rest(){
        double sum = 0;
        for (unsigned seg = 1; seg <= chanseg; seg++){
                for (unsigned t = this->routeSteps + 1; t < this->latq[seg].size(); t++){
                  sum += latq[seg][t];
                }
        }
        return sum;
}
```

C.4 Klasse RainGauges

(Code neu erstellt)

```
#include "StdAfx.h"
#include ".\raingauges.h"

//**** class by Uwe Hagenlocher****//
/* Format of theRainPos-file:
one header line with any or no content must be present. cols:
1. number of the station, can be any int and is not being used (starts with 0 inter-
nally)
2. x-position of that station (in Grid-cell-coordinates, lower left corner is 0,0)
3. y-position of that station
4. height of the station, put in anything if elevation-correction is not used
5. path + name of the file that contains data for that station, either full path
        or relative path to ZIN-project-folder

Format of the Data-files:
one header line with any or no content must be present. cols:
1. date in format d.M.YY or d.M.YYYY
2. consecutive minute of the day (0-1439)
3. data (decimal point like 1.3!)
No data: anything unreadable (e.g. strings or "1,5") or < 0

usage of this class:
1. construct a RainGauges-object
2. readDay when it is reached during the simulation
3. (set the idw-Grids for the whole day)
```

```
4. getDaySum if you need daily sums
5. (set idw-Grids for the present timestep)
6. getRainVal for the value of a specific cell and timestep*/

RainGauges::RainGauges(int methd, Grid* catchArea, Controller* rCont){
      cout << "setting up Rain-gauging net using ";
      this->rainCont       = rCont;
      this->method   = methd;
      this->xSize          = rainCont->getInt("xSize");
      this->ySize          = rainCont->getInt("ySize");
      this->ezgArea  = catchArea;
      this->noDatZero            = rainCont->getBool("NoData_zero");
      this->writeGrids     = rainCont->getBool("PCP_output");
      this->numberOfStats = 0;
      this->zinStep        = this->rainCont->getInt("ZINStep");
      this->mm_step        = this->rainCont->getBool("pcpUnit");
      this->reWriteSum     = this->rainCont->getBool("reWriteSums");
      this->sumIsNew             = false;
      this->numberZINSteps = 1440 / this->zinStep;
      this->dayRead        = this->yearRead = -1;
      this->distGrid             = 0;
      this->locSumGrid     = 0;
      this->distorder            = 0;
      this->stationGrid    = 0;
      this->dataGrid             = 0;
      this->demGrid        = 0;
      this->useDEM         = false;
      this->grad           = -1;
      this->refHeight = -1;
      int intDum;
      string readDummy;
      // open the file with positions and filenames for every rain-station:
      ifstream in(this->rainCont->getPath("RainPos").c_str());
      // find out how many Stations there are:
      while (getline(in, readDummy)){    //1. overread header line, 2. read up to eof
            in.clear();                                 // remove error flags
            in >> intDum;
            /* if the line starts with an int it contains station data -
            this allows for empty lines at the end, but not for text-lines!: */
            if (in)
              numberOfStats++;
      }
      in.clear();
      in.seekg(ios::beg);         // set reading position to the beginning
      getline(in, readDummy);     // overread the header line
      this->dataFiles = new string[numberOfStats];
      this->statPos  = new int*[numberOfStats];
      this->statVal  = new double*[numberOfStats];    //statVal[station][timestep]
precpitation value of station
      this->statLevel = new double[this->numberOfStats];
      for (unsigned i = 0; i < numberOfStats; i++){
            statPos[i] = new int[2];
            statVal[i] = new double[this->numberZINSteps];
            in >> intDum;               // overread the first col with the station
number
            in >> statPos[i][0];  // read the x-Pos          of Station i
            in >> statPos[i][1];  // read the y-Pos          of Station i
            in >> statLevel[i];         // read the elevation      of Station i
            in >> dataFiles[i];         // read the name of the related data-file
            getline(in, readDummy);     // ignore the rest of the line
            dataFiles[i] = this->rainCont->checkPath(dataFiles[i]);       // test if
the data-file exists
      }
      in.close();
      switch (this->method){
            case 0: // homogenous
            case 1: // Radar
              throw ReportException("voelliger Quark", 4);
              break;
            case 3:        // inverse distance weighting
              cout << "inverse distance weighting ... ";
              this->dataGrid = new Grid*[this->numberZINSteps];
```

```
            for (int i = 0; i < this->numberZINSteps; i++)
                  this->dataGrid[i] = 0;
            // no break! Thiessen-stuff is needed as well!
          case 2:        // Thiessen and inv dist
            if (method == 2)
                  cout << "Thiessen-Polygons ... ";
            // for the distances between one cell and every station:
            this->distGrid = new Grid*[this->numberOfStats];
            for (unsigned i = 0; i < this->numberOfStats; i++)
                  this->distGrid[i] = new Grid("double", this->rainCont);
            // for the station-ids in ascending order regarding the distance to one
cell:
            this->distorder = new int[this->numberOfStats];
            /* one Grid that contains the station-id of the nearest, one the second
            nearest etc., e.g. stationGrid[0] contains the id of the nearest station
for
            every raster cell: */
            this->stationGrid = new Grid*[this->numberOfStats];
            for (unsigned i = 0; i < numberOfStats; i++){
                  this->stationGrid[i]      = new Grid("int", this->rainCont);
                  this->distorder[i]        = i;   // initialize with any order
            }
            this->getDists();   // build distGrid and stationGrid
            break;        // end Thiessen (& inv. Dist.)
          default:
            throw ReportException("Rain method-code not valid!", 4);
      } // end switch
      cout << "\tdone!" << endl;
      /*if (this->writeGrids)
            this->testOutput();*/
}

RainGauges::~RainGauges(){
      delete [] distorder;
      distorder = 0;
      for (unsigned i = 0; i < numberOfStats; i++){
            if (statPos){
            delete [] statPos[i];
            statPos[i] = 0;
            }
            if (statVal){
              delete [] statVal[i];
              statVal[i] = 0;
            }
      }
      delete [] statPos;
      statPos = 0;
      delete [] statVal;
      statVal = 0;
      delete [] dataFiles;
      dataFiles = 0;
      delete [] statLevel;
      statLevel = 0;
      if (this->dataGrid != 0){
            for (int i = 0; i < numberZINSteps; i++){
              delete this->dataGrid[i];
              this->dataGrid[i] = 0;
            }
            delete [] this->dataGrid;
            dataGrid = 0;
      }
      if (this->distGrid != 0){
            for (unsigned i = 0; i < numberOfStats; i++){
              delete this->distGrid[i];
              this->distGrid[i] = 0;
            }
            delete [] this->distGrid;
            distGrid = 0;
      }
      if (this->stationGrid != 0){
            for (unsigned i = 0; i < numberOfStats; i++){
              delete this->stationGrid[i];
```

```
            this->stationGrid[i] = 0;
          }
          delete [] this->stationGrid;
          stationGrid = 0;
    }
}

void RainGauges::getDists(){
    /*    1. fill distGrid[i] with the distance to station i for every cell
          2. fill stationGrid[i] with the station-id of the i-nearest station */
    int    swappee;             // schnapp, schnapp, schnappi!!!
    double* distVal;            // distance of the current cell to the index-station
    distVal = new double[this->numberOfStats];
    bool done;                  // sorting finished?
    for (int y = 0; y < ySize; y++){
          for (int x = 0; x < xSize; x++){
            for (unsigned i = 0; i < this->numberOfStats; i++){
                  // distance of Grid-cell x,y to station i (Pythagoras):
                  distVal[i] = sqrt(pow(double(statPos[i][0] - x), 2.0) +
pow(double(statPos[i][1] - y), 2.0));
                  this->distGrid[i]->putDoubleAt(x, y, distVal[i]);
            }
            /* Bubble-sort the stations by their distance to the cell
            and store their order in "distorder":
            (bubble-sort should be efficient here as in many cases the
            stations are pre-sorted from the preceeding neighbouring cell!)*/
            done = false;
            while (!done){
                  done = true;
                  for (unsigned i = 0; i + 1 < this->numberOfStats; i++)
                        if (distVal[distorder[i]] > distVal[distorder[i+1]]){
                              done                  = false;
                              swappee                     = distorder[i];
                              distorder[i]  = distorder[i + 1];
                              distorder[i + 1]= swappee;
                        }
            }     // end sorting
            for (unsigned i = 0; i < numberOfStats; i++){
                  // write the distance-ranks to the Grid
                  if (this->ezgArea->getIntegerAt(x, y) == -9999)
                        stationGrid[i]->putIntegerAt(x, y, -9999);
                  else
                        stationGrid[i]->putIntegerAt(x, y, this->distorder[i]);
            }
          }        // end x
    }     // end y
    delete [] distVal;
    distVal = 0;
}

double RainGauges::getRainVal(int x, int y, int timestep){ // timestep: 1..n
    /* get the precipitation-value for one cell, one timestep
    (timestep-1 because the Execution-steps start with 1, here 0) */
    unsigned prox;
    double val;
    timestep--;
    switch (this->method){
          case 2:        // Thiessen
            int stat;
            prox = 0;
            do { // look for the nearest station that has a value and return it
                  if (prox == 1 && noDatZero)
                        return 0;
                  stat = this->stationGrid[prox]->getIntegerAt(x, y);
                  val = this->statVal[stat][timestep];
                  prox++;
            } while (val < -9990 && prox < this->numberOfStats);
            break;
          case 3: // inverse Distance
            val = this->dataGrid[timestep]->getDoubleAt(x, y);
            if (timestep > lastStep && timestep > 1){ // free precious memory, but
don´t delete Grid[0]!
```

```
                        delete this->dataGrid[timestep - 1];
                        this->dataGrid[timestep - 1] = 0;
                        this->lastStep = timestep;
                }
                break;
              default:
                throw ReportException("Rain method-code not valid!", 4);
        }
        return 0 > val? 0 : val;
}

void RainGauges::readDay(int yyyy, int dayOYear, int* dryStepsInHour, Grid* heights){
        /* this readDay is used if you chose to use a DEM for correcting precipitation
with height.
        It makes the DEM accessible, sets useDEM = true and checks if the levels in the
DEM are
        consistent with those from the RainPos file. Then it hands over the paramters to
        readDay(int, int) to do the work.
        Note: Do not change the Grid that heights/demGrid points to, its your DEM from
TRAIN!!!*/
        double elev;
        string  reader;
        this->demGrid  = heights;
        this->useDEM   = true;
        for (unsigned i = 0; i < this->numberOfStats; i++){
                // check if the levels in the DEM are consistent with those from the Rain-
Pos file
                // don´t check if the station is outside dem-rectangle:
                if (statPos[i][0] < 0 || statPos[i][0] >= this->xSize ||
                  statPos[i][1] < 0 || statPos[i][1] >= this->ySize)
                  continue;
                elev = this->demGrid->getFloatAt(statPos[i][0], statPos[i][1]);
                if (elev < -9990)
                  continue;     // no data available from the DEM, so try the next statio-
n[i]
                //compare the distance within a tolerance:
                if (abs(elev - this->statLevel[i]) > 0.5){      // oho! here you go, DEM
and file differ!
                  cout << "value of elevation of station " << i + 1 << " read from the
\"RainPos\"-file is ";
                  cout << this->statLevel[i] << endl;
                  cout << "value of the DEM at that position is " << elev;
                  do{           // try reading a value until DAU enters a double:
                        cout  << endl << "enter the correct value: ";
                        cin.clear();
                        cin.sync();         // forget any previously read wrong entries
                  }while(!(cin >> statLevel[i]));
                }
        }
        this->readDay(yyyy, dayOYear, dryStepsInHour);
}

void RainGauges::setNoData(double* vals){
        for (int i = 0; i < this->numberZINSteps; i++){
                vals[i] = -9999.0;
        }
}

void RainGauges::readDay(int yyyy, int dayOYear, int* dryStepsInHour){
        /* read the pcp-data of the current day (dayOYear), all methods
        the aim is to assign a value to statVal that is either
        read from a file or -9999 if noData is present
        also drySteps in hour is set */
        this->yearRead       = yyyy;
        this->dayRead  = dayOYear;
        lastStep = -1;
        ifstream in;
        int     intRead;
        double  dRead;
        string  strDum;
        string  dateStr = this->rainCont->getDateString(dayOYear, yyyy);
        string  dateRead;
```

```
      string  lastDate;
      bool  noData = false;
      bool  anyData = false;
      bool  done;
      cout << "reading Precipitation Data ...\n";
      for (unsigned i = 0; i < this->numberOfStats; i++){
            // looping through all stations
            noData = false;
            in.open(this->dataFiles[i].c_str());
            if (!in)
              throw ReportException(dataFiles[i], 1);
            cout << "reading " + dataFiles[i] + " ... ";
            getline(in, strDum);         // header line
            done = false;
            while (!done && in >> dateRead){   // search the date in the file
              if (dateRead != lastDate){
                    done = !(this->rainCont->dateCompare(dateRead, dateStr) < 0);
                    if (done)
                          break;         // found date!
                    lastDate = dateRead;
              }
              getline(in, strDum);       // eat rest of the line
            }
            if (this->rainCont->dateCompare(dateRead, dateStr) > 0 || !in){
              // file starts after or ends before the desired year
              setNoData(statVal[i]);
              in.clear();
              in.close();
              cout << "\t!!!date not found!!!\n";
              continue;
            }
            for (int miod = 0; miod < 1440; miod += this->zinStep){
              /* for the current station loop through the day (miod: minute of the
day)
              and read the data */
              in >> intRead;             // read minute
              if (intRead != miod)                    // oops! wrong time!
                    throw ReportException(dataFiles[i] + " minute!", 2);
              in >> dRead;  // read data
              if (this->mm_step)
                    dRead *= 60/this->zinStep;
              if (dRead < 0)      // no Data (indicated by negative value)
                    dRead = -9999;
              if (!in){                  // no Data (indicated by string value)
                    dRead = -9999;
                    in.clear();  // remove error flag
                    getline(in, strDum);       // get rid of the rest of the line
              }
              if (!anyData && dRead > -9990)    // ok data
                    anyData = true;
              statVal[i][miod / this->zinStep] = dRead;             // store data
              if (!in.eof() && miod < 1440 - this->zinStep){
                    /* not if the model run is to end in the middle of the day
                    or the last timestep is reached*/
                    in >> dateRead;      // read date
                    if (this->rainCont->dateCompare(dateRead, dateStr)) // oops! wrong
date!
                          throw ReportException(dataFiles[i] + " date!", 2);
              }
            }
            in.close();
            if (anyData)
              cout << "\tdone!\n";
            else
              cout << "\tno Data in period!\n";
      }
      for (int steps = 0; steps < this->numberZINSteps; steps++)
            /* initialize dryStepsInHour with the maximum possible value which is the
number of
            timesteps per hour: */
            dryStepsInHour[int(steps * this->zinStep / 60) + 1] = 60 / this->zinStep;
      for (int steps = 0; steps < this->numberZINSteps; steps++){
```

```
            for (unsigned i = 0; i < this->numberOfStats; i++){
              // if you come across the first rain at any one station:
              if (statVal[i][steps] > 0){
                    dryStepsInHour[int(steps * this->zinStep / 60) + 1]--;
                    break;        // no need to look at other stations, so look at the
next timestep
              }
            }
      }
      cout << "... pcp-reading done!" << endl;
}

void RainGauges::setGridStep(int step){ // step is from 0..n-1
      /* calculate the inverse-distance-weighted precipitation-value and store it
      in dataGrid[step]
      (1)   P(x,y) = a * sum(P_i/distance^2)   (sum: over all stations(i))
      (2) a = 1/sum(1/distance^2)                              (sum: over all
stations(i))
      where
      P(x,y): Precipitation at (x,y)
      P_i: Precipitation at station i
      distance: distance between this cell and station i*/
      unsigned use_Stat;
      unsigned distInd;
      unsigned maxStats = this->rainCont->getInt("Max_Stats");
      unsigned failed;
      unsigned totRuns;
      unsigned pos = step;
      if (!this->readDaily)
            pos = 0;
      double  a;
      double  pDivA;                    // result of the sum-part of equation (1)
      double  result;
      double  cellLevel = 0.0;
      double  val;
      double  dist;
      double  distSquare;
      if (!this->dataGrid[pos])
            this->dataGrid[pos] = new Grid("double", this->rainCont);
      vector<unsigned> goodStats;       // contains marks whether the station contains
data in the present timestep
      goodStats.resize(this->numberOfStats);
      vector<unsigned> nearestStats;    // contains the index of the maxStats nearest
stations to one cell
      // end output variables
      if (this->useDEM){
            this->grad           = this->rainCont->getFloat("rainGrad") / 10000;
      // conversion from %/100m to 1/m
            this->refHeight = this->rainCont->getInt("refHeight");
      }
      failed = 0;
      // check which stations contain data and can be used:
      for (unsigned e = 0; e < this->numberOfStats; e++){
            if (this->statVal[e][step] >= 0)   // if station e contains data at that
                                               timestep...
              goodStats[e] = 1;                // mark that station as "good" (1)
            else{
              goodStats[e] = 0;                // else mark that station as "bad" (0)
              failed++;           // remember how many stations did not contain data
            }
      }
      // use the maxStats nearest stations for idw, or as many as do have good data,
whichever is less:
      totRuns = min(numberOfStats - failed, maxStats);
      nearestStats.resize(totRuns);
      for (int y = 0; y < this->ySize; y++){
            for (int x = 0; x < this->xSize; x++){
              cellLevel = this->demGrid->getFloatAt(x, y);
              if (cellLevel < -9990){ // cell is outside the catchment, so proceed
with next x
                    this->dataGrid[pos]->putDoubleAt(x, y, -9999);
                    if (this->locSumGrid)
```

```
                        this->locSumGrid->putDoubleAt(x, y, -9999);
                  continue;
            }
            a = pDivA = 0.0;
            failed = 0;
            /* find the maxStats number of nearest Stations and assign their indices
to nearestStats:
            (loop as long as >>the number of stations to be used (totRuns) minus the
number of
            stations allready processed (but only if they belong to the goodStats,
thus the "- failed")<<
            i.e., is not 0)*/
            for (unsigned i = 0; totRuns - (i - failed); i++){
                  distInd = stationGrid[i]->getIntegerAt(x, y);  // index of station
i-closest to this cell
                  if (goodStats[distInd]){                                    //
Station marked as good?
                        nearestStats[i - failed] = distInd;     // Stat-indices in
order of distance, offset by failed
                  }
                  else failed++;
           // no data at that station
            }
            for (unsigned i = 0; i < totRuns; i++){
                  use_Stat = nearestStats[i];
                  dist   = this->distGrid[use_Stat]->getDoubleAt(x, y);
                  val          = this->statVal[use_Stat][step];
                  if (this->useDEM){
                        /* reducing the value according to the level of the stati-
on: */
                        val /= 1 + this->grad * (this->statLevel[use_Stat] - this-
>refHeight);
                  }
                  if (dist == 0){ // avoid division by 0 for the cell that lies at
the gauge
                        a = 1;
                        pDivA = val;
                        break;
                  }
                  distSquare = dist * dist;
                  a += 1 / distSquare;       // this is really accumulating 1/a!!
                  pDivA += val / distSquare;
            }
            a = 1 / a;          // now a really is a!
            result = a * pDivA;
            if (this->useDEM)
                  /* lifting the value back up according to the level of the raster
cell: */
                  result *= 1 + this->grad * (cellLevel - this->refHeight);
            this->dataGrid[pos]->putDoubleAt(x, y, result);
            if (result > 0 && this->locSumGrid)
                  this->locSumGrid->putDoubleAt(x, y, this->locSumGrid->getDouble-
At(x, y) + result / 60 * this->zinStep);
           } // end x
      } // end y
      if (this->writeGrids){
            ostringstream conv(ios::in);
            string outname;
            // for output of every Grid construct the folder-name:
            conv.str("");
            conv << this->yearRead << "_" << this->dayRead << "_";
            conv << int (this->zinStep * step / 60) << "_";
            conv << step - int (this->zinStep * step / 60) * 60;
            outname = this->rainCont->getPath("Output") + "Raingrids\\" + conv.str();
            this->dataGrid[pos]->writeGrid(outname);
      }
}

void RainGauges::setGridDay(){
      cout << "calculating inverse-distance-Grids ...\ntimestep:";
      this->locSumGrid = new Grid("double", this->rainCont);
      this->locSumGrid->setValueD(0.0);
```

```
	for (int z = 1; z <= numberZINSteps; z *= 10)
		cout << " ";
	for (int step = 0; step < this->numberZINSteps; step++){
		// screen-output of present timestep:
		for (int z = 1; z <= step; z *= 10)
		  cout << "\b";
		cout << step + 1;
		this->setGridStep(step);
	} // end step
	cout << "\n";
	ostringstream conv;
	string filename = this->rainCont->getPath("Outfold_sums");
	conv.str("");
	conv << this->yearRead << "_" << this->dayRead;
	filename += conv.str();
	if (!this->reWriteSum){
		ifstream tester(filename.c_str());
		if (!tester){
		  tester.clear();
		  this->locSumGrid->writeGrid(filename);
		}else
		  tester.close();
	}else
		this->locSumGrid->writeGrid(filename);
	this->sumIsNew = true;
}

bool RainGauges::getDaySum(int yyyy, int dayOYear, Grid& daySumGrid){
	// read the daily sum into Execution::inputGrid (pointed at by daySumGrid)
	double  val;
	ostringstream conv(ios::in);
	string filename;
	ifstream test;
	switch (this->method){
		case 2: // Thiessen
		  for (int y = 0; y < ySize; y++){
			for (int x = 0; x < xSize; x++){
				val = 0;
				if (this->ezgArea->getIntegerAt(x, y) < -9990){
					daySumGrid.putDoubleAt(x, y, -9999.0);
					continue;
				}
				for (int i = 0; i < 1440; i += this->zinStep){
					// +1: getRainVal starts with timestep 1
					val += max(this->getRainVal(x, y, i/zinStep +
1)/60*this->zinStep, 0.0);// conversion from mm/h in mm
				}
				daySumGrid.putDoubleAt(x, y, val);
			}
		  }
		  break;
		case 3: //idw
		  // check if the daily-sum file alredy exists:
		  filename = this->rainCont->getPath("Outfold_sums");
		  conv << yyyy << "_" << dayOYear;
		  filename += conv.str() + ".asc";
		  test.open(filename.c_str());
		  if (!test || (this->reWriteSum && !this->sumIsNew))
			return false;
		  this->sumIsNew = false;
		  if (this->locSumGrid){
			for (int y = 0; y < ySize; y++){
				for (int x = 0; x < xSize; x++){
					val = 0.0;
					if (this->ezgArea->getIntegerAt(x, y) < -9990){
						daySumGrid.putDoubleAt(x, y, -9999.0);
					}
					else
						daySumGrid.putDoubleAt(x, y, this->locSumGrid-
>getDoubleAt(x, y)/60*this->zinStep);
				}
			}
```

```
            }else
                  daySumGrid = Grid(filename, "double", this->rainCont);
            delete this->locSumGrid;
            this->locSumGrid = 0;
            break;
          default:
            throw ReportException("Rain method-code not valid!", 4);
      }
      return true;
}
```

C.5 Klasse Routing

(Ausschnitt: Schleife, die die Muskingum-Cunge-Iteration enthält)

```
/* estimation value for q (Q_ref or qber), arithmetic mean of
- inflow from upstream (this timestep),
- inflow from upstream (last timestep) and
- q of this segement, last timestep without lateral inflow.
This is just a starting value for the calculation of q, any value > 0 should
do, but this makes the following loop a little faster: */
qber[ic] = (q[upper[ic]] + q[trib1[ic]] + q[trib2[ic]]
      + last_q[upper[ic]] + last_q[trib1[ic]] + last_q[trib2[ic]]
      + (last_q[ic] - last_lat[ic])) / 3;
/*stepwise approximation of q and qber, following
the non-linear Muskingum-Cunge scheme:*/
iterations = 0;
optErrCurr = OptErr;
do {
      iterations++;
      if (qber[ic] <= 1e-200){
            /* depth is calculated as log(qber ...), so no 0 please!
            if iterations == 1: all inflows and last_q are 0, so there can´t be any
runoff.
            if iterations  > 1: qber was > 0, but now q converges to 0 or negative
value */
            q[ic] = 0;
            break;
      }
      else{ //calculation of parameters for Muskingum Routing
            if (sediment){
              cvelo[ic] = velocity[time][ic];
            }
            else{
              if (chantype[ic] == 1){    // pipes
                    depth[ic] = this->depthPipes(qber[ic], depth_full_type[1], n_man-
n[ic], s[ic], ic);
              }
              else{                                 //open channels
                    depth[ic] = exp(0.6 * log(((qber[ic] / stepSeconds) * n_mann[ic])
/
                          (sqrt(s[ic]) * b_var[ic])));
                    velo[ic] = qber[ic] / (stepSeconds * b_var[ic] * depth[ic]);
                    cvelo[ic] = 5.0 / 3.0 * velo[ic];
              }
            }
            kmus [ic] = lengdi[ic] / cvelo[ic];
            // division by stepSeconds: conversion of qber from m³/step -> m³/s:
            xmus[ic] = 0.5 - (qber[ic] / (stepSeconds * 2 * cvelo[ic] * b_var[ic] *
s[ic] * lengdi[ic]));
      }
      // stepSeconds == timestep * 60 == timestep in seconds
      c0[ic] = 2 * kmus[ic] * (1 - xmus[ic]) + stepSeconds;
      c1[ic] = (stepSeconds - (2 * kmus[ic] * xmus[ic])) / c0[ic];
      c2[ic] = (stepSeconds + (2 * kmus[ic] * xmus[ic])) / c0[ic];
      c3[ic] = (2 * kmus[ic] * (1 - xmus[ic]) - stepSeconds) / c0[ic];
      /* calculation of q: */
      q[ic] = c1[ic] * (upper_q[ic] + qtrib1[ic] + qtrib2[ic])
            + c2[ic] * (upperLast_q[ic] + last_trib1[ic] + last_trib2[ic])
            + c3[ic] * (last_q[ic] - last_lat[ic]);
```

```
            //+ c4[ic] * latq[upper[ic]];
      maxIt = max(iterations, maxIt);
      if (iterations >= 500){     //q and qber do not seem to converge, so increase
the acceptable error
            iterations = 0;
            optErrCurr *= 1.1;
      }
}while (abs(qber[ic] - q[ic]) > abs(q[ic] * optErrCurr) && (qber[ic] = q[ic]));
/* the while(...) condition:
first part: if |q - qber| > |q * err|: true
second part: this IS supposed to be an assignment rather than a comparison!
note that it will be executed _after_ the first part which will not be affected by it!
it yields the value of q which will be interpreted as "true" for all but q == 0.0 */
//lateral inflow of the Q-concentration is added at the downstream end of the segment
after routing:
q[ic] += latq[ic];
```

Ehrenwörtliche Erklärung

Hiermit erkläre ich, dass ich die Arbeit selbstständig und nur unter Verwendung der angegebenen Hilfsmittel angefertigt habe.

Freiburg, 20. März 2008 (Uwe Hagenlocher)