Formulation and Testing of a Distributed Triangular Irregular Network Rainfall/Runoff Model

by

Scott Michael Rybarczyk

Submitted to the Department of Civil and Environmental Engineering
in partial fulfillment of the requirements for the degree of

Master of Science in Civil and Environmental Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2000

Author .....
Department of Civil and Environmental Engineering
August 11th, 2000

Certified by .....
Rafael L. Bras
Professor
Thesis Supervisor

Accepted by .....
Daniele Veneziano
Chairman, Department Committee on Graduate Students

Formulation and Testing of a Distributed Triangular Irregular Network Rainfall/Runoff Model

by

Scott Michael Rybarczyk

Submitted to the Department of Civil and Environmental Engineering
on August 11th, 2000, in partial fulfillment of the
requirements for the degree of
Master of Science in Civil and Environmental Engineering

Abstract

In this thesis, a new distributed, continuous simulation model is developed for flood forecasting. This new model, tRIBS (triangulated Real-Time Interactive Basin Simulator), is created by integrating two models previously developed. A landscape evolution model, CHILD, is used to create the triangular irregular network (TIN) of tRIBS while the original RIBS model is used to provide runoff and saturated/unsaturated groundwater dynamics in the system.

These two base models of tRIBS are described and the modifications to CHILD and RIBS are presented. The CHILD model is modified to accept time varying distributed inputs and a saturated zone groundwater flow routine is created. The RIBS model is modified to allow for continuous simulation and capillary suction.

This thesis also develops the datasets needed for the tRIBS model. Starting from Digital Elevation Models, watersheds are delineated and then manipulated using geographic information systems to form a TIN. Algorithms to create distributed rainfall inputs and stream channels are also developed for use in the tRIBS model.

With the model and dataset completed, the model is sucessfully tested and calibrated to the Peacheater Creek watershed. Results are very promising.

Thesis Supervisor: Rafael L. Bras
Title: Professor

Acknowledgments

To avoid any semblance of favoritism, all these acknowledgments are in reverse chronological order. First, thanks go out to all the folks at the NASA Land Surface Hydrology Program, MIT, NSF, and the National Weather Service for funding me while I completed this thesis. I didn't live like a king, but I avoided the huge loans that sometimes accompany graduate school. Specifically, thanks go out to Michael Smith, Victor Koren, Seann Reed and Robert Shedd for all their assistance in providing data and calibrated Sacramento model runs which were crucial to my success.

Everyone at MIT has really been wonderful in helping me with my work and my sanity. Advisor Rafael Bras and Valeri Ivanov, deserve extra special kudos for their assistance. Greg Tucker, Nicole Gasparini, Daniel Collins, Jean Fitzmaurice, Babar Bhatti, Holly Michael, Brian Crounse, Sheila Frankel, Megan Kogut, Luis Perez–Prado and Elaine Healy among many others saved me many times by listening to me rant or taking me out for a 'liquid beverage' when needed.

Other friends in Boston have been a great addition to my life. Where would I be without the Tang trio of Arrin, Pat, and Chip (Shopping carts, anyone?) or everyone in the Inline Club of Boston whom I miss already. To Kim, Mitch, Mark, Lori and numerous others, I promise to visit and skate with you again soon.

To all the people back in Buffalo who have known me forever, you deserve a mention here as well. To Joe DePinto and Ralph Rumer, thanks for giving a UB undergraduate the chance to thrive. To all my friends in West Seneca who grew up with me, especially Eric Nalewajek and Brian Kistner, I owe you for being there through good and bad.

Finally, I would like to acknowledge those who have known me since birth. My family deserves a huge deal of credit for who I am. To the many aunts, uncles, cousins, grandparents, and everyone else I say thanks. To my mother and my baby sister, Lisa, that thanks is multiplied a thousandfold.

Lastly, this document is dedicated in the memory of my father, Kenneth T. Rybarczyk. You would have been so proud.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Distributed data sources have created a quandary for flood forecasters; how can one best use this wealth of information? Traditionally, flood forecasting has relied on lumped rainfall/runoff models. These models take all available data for a watershed and reduce it to a single point. This type of modeling has been done for decades, and is still being used by many major flood forecasting agencies, including the US National Weather Service. Recently, researchers have been focusing on distributed hydrologic modeling, as it incorporates this increased information and begins to look at the process physics behind rainfall/runoff modeling. The question that has yet to be answered is the following: "Do distributed models perform better than the lumped rainfall/runoff models?" This project will take the initial steps needed for answering this question. A rainfall/runoff model will be modified to create a future platform for this research and the large volumes of data needed for a distributed model will be collected and manipulated to support the modeling effort.

At this time, flood forecasting in the United States and most of the world relies on lumped parametric models [16]. When these models were first created, distributed data did not exist and computer power was extremely limited. Detailed paper maps of topography did exist, but transforming that information into a digital format was nearly impossible until the advent of Geographic Information Systems (GIS) in the

late 1970's. Rainfall was measured using gage data as no operational radars were available at the time. With this scarcity of data and lack of computer power, a lumped model which represents an entire basin as one point was reasonable. Also, when calibrated properly, these models reproduce observed discharges quite well with only moderate computational effort.

Nevertheless, these rainfall runoff models have significant problems. Lumped flood forecasting models ignore the topography of a region so two basins with radically different shapes and slopes, but the same area, look alike to the model. Precipitation input from rain gages or radar is averaged over the entire basin, so a storm that rains on the upland area of a watershed produces the same output as a storm that occurs near the outlet. Any other distributed data (i.e. soils data) is ignored by these models. These systems represent signals, not physics when trying to model hydrologic response in a watershed. Also, only the outlet of a watershed is modeled by these systems; information on the behavior internal to the basin outlet cannot be determined by a lumped system.

Distributed hydrologic models have some significant issues as well. The largest problem is computational demand. In a distributed model, a watershed is defined by thousands of elements, instead of the single element used in a lumped model. Increased computer power has made this a lesser issue, but it is still significant. Moving from a single element in a lumped model to thousands of elements in a distributed model has serious implications when calibrating the model. In 1992, Keith Beven and Andrew Binley [7] questioned whether distributed models could be robustly calibrated. No definitive answer has yet been given. In addition, it has been shown that the large number of parameters present in these models make them susceptible to non-unique calibrations. Two very different parameterizations can give the same results [63]. Also, good calibration can occur with unacceptable watershed physics [30]. These difficulties in calibration and parameterization can even make these models impractical in use as operational flood forecasting models [60].

This work hypothesizes that the proper identification of the two major inputs to hydrologic modeling (precipitation and topography) will be the dominant factor in

determining the hydrologic output of the model. While still important, the calibration of distributed soil parameters have much less impact on the model results, allowing this type of modeling work to continue.

The spatial and temporal resolution of data sets continues to increase. Two recent examples in topography and rainfall measurement show this trend in distributed data. Distributed rainfall data for the TRMM experiment is providing new and exciting images of rainfall, showing details that cannot be seen by ground radar [65]. At this point, the TRMM system only measures rainfall over the tropics with low temporal resolution. Nevertheless, there are serious discussions about a future global precipitation mission with an increased temporal scale. Advances in distributed topography can best be seen in the recent Shuttle Radar Topography Mission (SRTM). SRTM measured topography around the world at a 30m resolution. The accuracy should also be quite good with multiple images in most areas from 60°N to 60°S latitude.

1.2 A Brief History of Flood Forecasting Models

Beginning with the work of Mulvaney in 1851 [49] and continuing through the present, numerous models have been created in an attempt to forecast floods. An attempt to look at all of the models created for this purpose would be impossible, and this work will only look at a small cross section of the models developed. The focus will be on models which represent a significant change in the methodology of distributed hydrologic modeling along with models that are widely used in operational flood forecasting and engineering design. This history will be split into two subsections, lumped and distributed models, each organized chronologically.

1.2.1 Lumped Hydrologic Models

The earliest work in flood forecasting began with the use of so called 'black-box' models. These models use a statistical approach to find the output of a catchment. By matching the input and output using only mathematics, no physically based function is required. These models work well within the data ranges set by the historical data

used to create the black-box model. Unfortunately, the extreme events outside the historical data limits are often of great interest to a flood forecaster and extrapolating to get these forecasts works poorly. Commonly, there is inherent linearity in these models, and the extrapolations are often of little use [4]. Models which fit the black box mode include those using frequency analysis and the unit hydrograph approach.

In 1913, Horton [35] began using frequency analysis to predict the occurence of floods. Using a normal distribution, Horton was able to apply frequency analysis to flood forecasting for the first time. Others followed this same approach using different distributions including lognormal [34] and the skewed Pearson distribution. [23] Frequency analysis is still used today and the log Pearson Type III distribution is the most widely used due to its endorsement by various federal agencies [76].

Flood Frequency Analysis can calculate the probability that a flood of a specific magnitude will occur, but it does not model a specific event. The unit hydrograph developed by Sherman in 1933 [64] can forecast streamflow based upon specific rainfall events. Defined as the response to a unit depth of rainfall uniformly distributed over an entire catchment for a specific length of time, a unit hydrograph can be calculated directly from rainfall/runoff data. Alternatively, a synthetic unit hydrograph can be used to find the unit hydrograph indirectly in an ungaged watershed. Using superposition, unit hydrographs can then be combined to get estimates of what will happen during future storms including those with time varying rainfall events.

One of the major assumptions in the unit hydrograph method is that watershed response acts in a linear manner. Due to the fact that discharge and velocity are nonlinear functions of stage and flow, this assumption is not always valid. To correct for this, nonlinear versions of the unit hydrograph were developed in the 1960's [3]. The unit hydrograph approach also evolved into the instantaneous unit hydrograph which was used in the used in many flood forecasting models in the 1950's including the Nash [50] and Dooge [19] storm response models. Further refinement of the unit hydrograph using characteristics of the basin structure allowed for the creation of a geomorphologic unit hydrograph ([61],[33]).

All of the models presented previously are run for single storm events. An early

and important example of a lumped, continuously simulated, conceptual model was the Stanford Watershed Model (SWM) [44] created in 1960. One of the first continuous, conceptual models, SWM served as the basic platform for many future models including the widely used HSPF (Hydrologic Simulation Program-Fortran) model. In synthesizing daily or hourly streamflow, SWM uses both precipitation and evaporation to drive the model. Other meterological data is included to calculate snowmelt (if needed). Water is stored in three boxes in SWM including upper storage, lower storage, and groundwater storage. The upper zone and lower zone storage both account for infiltration, interflow and inflow to the groundwater. The major difference between the two is the reaction to rainfall events. The upper zone storage is active and produces runoff from minor storms, while the lower zone storage becomes active and produces runoff in major storms exclusively.

Other significant conceptual, lumped, hydrologic models were also developed shortly after SWM. One of these was the HEC-1 (Hydrologic Engineering Center) Flood Hydrograph Package [54]. Created in 1968 and still in use in today, HEC-1 is an event based model which calculates discharge at the outlet. This model uses a basin approach, where each basin is modeled using a series of interconnected subbasins each with averaged hydrologic parameters. In 1970, SWMM (Stormwater Management Model) was developed [47]. SWMM includes all the basics of a event based, conceptual model with added functionality to model water quality in these discharges. The continuously simulated Sacramento Model, first developed as the Generalized Streamflow Simulation System in 1973 [11], can be considered in this same category as well. The Sacramento Model will be described in further detail in Chapter 2.

In each of the previous models, the runoff was controlled by the soil surface. This follows the work of Horton ([36],[37]), where runoff occurred due to infiltration excess. If rainfall exceeds the capacity of the soil at the surface, runoff occurs. This hortonian runoff can produce an adequate result for some watersheds, but it neglects other forms of runoff due to subsurface exfiltration and return flow [20]. A variable source area concept was developed to account for the runoff contribution of saturated areas along streams and rivers. A semi-distributed model developed by Beven and Kirkby [8]

began looking at these processes by predicting the storage in soils based upon the contributing area and topographic structure of a basin. This approach, often referred to as the TOPMODEL approach, has been used to define similar concepts found in the RIBS (Realtime Interactive Basin Simulator) distributed model detailed further in Chapter 2.

1.2.2 Distributed Hydrologic Models

An early definition of a physically based distributed hydrologic model was given in a paper by Freeze and Harlan in 1969 [24]. Freeze and Harlan presented the specifications for a three dimensional model with complete analytic expressions defining the hydrological processes. For the last thirty years, various researchers have taken this framework and simplified it for practical use in flood forecasting. The three dimensional model domain has been simplified to two or even one dimensions, and the nonlinear partial differential equations have been replaced with numerical solutions. Papers detailing individual concepts in distributed modeling (groundwater flows, unsaturated soil water flow, and channel routing) are common, but only in the last ten to fifteen years have complete distributed hydrologic models entered the scene on a reasonable catchment scale.

One example of a complete distributed hydrologic model is the Systeme Hydrologique Europeen (SHE) ([1],[2]). The basic SHE model was completed in 1982 with the joint effort of researchers from the Danish Hydraulic Institute, the Institute of Hydrology (UK), and SOGREAH (France) [5]. SHE is a basin scale model, simulating overland flow, channel flow, saturated and unsaturated subsurface flow, along with interception, evaporation, and snowmelt. Hydrologic processes in the SHE model are represented in either one or two dimensions. Channel flow and unsaturated zone subsurface flow are averaged to a single dimension. Overland flow and saturated zone subsurface flow are calculated in two dimensions. These assumptions produce unsaturated zone flow in the vertical direction and saturated zone flow in the horizontal direction. The finite difference method is used to calculate mass and energy conservation in the SHE model and the kinematic wave approximation is used to find

surface flow in the model. This model has went through many iterations and is now being used in sediment transport modeling as well [78].

Following SHE is the CASC2D (A Cascade of Planes in Two Dimensions) Model. Starting as a overland flow algorithm in 1991, CASC2D was modified in 1995 to create a physically based hydrologic model for flood forecasting [42]. Later modifications in 1997 allowed CASC2D to run continuously [56]. Developed at Colorado State University for the US Army, the model is a raster based, infiltration excess (hortonian), hydrologic model. It includes continuous soil moisture accounting, interception, infiltration, surface and channel routing along with sediment transport [55]. The dimensionality of the model is similar to that of SHE with two dimensional overland flow and one dimensional flow through the unsaturated zone. Soil infiltration is calculated through a Green and Ampt approach [31] and the soil column is assumed to be infinitely deep. This assumption will not be adequate in situations where the water table level plays an important part. The calibration of CASC2D needs years of data in continuous mode with a full set of meterological variables including evapotranspiration.

1.3 Project Approach

The first major effort in this project includes the modification of RIBS (Realtime Integrated Basin Simulator), an existing distributed hydrologic model. RIBS will be modified by altering both the model physics and the model framework. The model physics will be changed by moving RIBS from an event based to a continuous model. This includes the addition of interstorm conditions (to allow for evaporation) and a complete groundwater system including both saturated and unsaturated zone components. The framework of the RIBS model is modified to allow for operation on an irregular mesh. This was completed by modifying RIBS to fit within the TIN (Triangular Irregular Mesh) framework of the CHILD (Channel-Hillslope Integrated Landscape Development) Model. This integration allows for variable resolution in the new RIBS model and provides for future integration of the two models. This process

of moving towards a new version of RIBS is presented in two chapters. Chapter 2 describes the original RIBS model along with the framework of the CHILD model. The lumped Sacramento-SMA model used in Chapter 5 is also briefly described here. A discussion of the changes made to the RIBS and CHILD models is given in Chapter 3.

The distributed data used in a rainfall/runoff model are an essential part of the modeling effort. In chapter 4, the process used to retrieve and manipulate the various data sets used in this project are given. This was the second major effort of this project, and a large period of time was spent working with distributed data. All of this colleted data has been published on the web to give others access to this data. With the data collection complete, the RIBS and Sacramento-SMA models are calibrated and run for a basin in Oklahoma and the results are shown as Chapter 5. Chapter 6 then presents the final conclusions of this work.

Chapter 2

The Models

2.1 RIBS

The following is a detailed description of the RIBS model. In completing this description, the style and structure of the publications by Garrote and Cabral ([26], [12], [27]) detailing the original construction of RIBS have influenced this section of the report significantly. RIBS was first completed in 1993 as an event based distributed flood forecasting system built on a raster (grid) framework. A general framework of RIBS is given in Figure 2-1. As seen in Figure 2-1, the model begins with a set of initialized basin properties. These include water table depth, soil characteristics, and topographic features of the watershed (slopes, elevations, distance to stream and outlet). These initial basin properties are combined with forecasted or measured rainfall at each time step and input to the rainfall/runoff transformation model. The rainfall/runoff transformation model then produces stream hydrographs and various internal variable states including moisture content and runoff generation. All of these inputs and outputs can be viewed and edited using the RIBS user interface.

2.1.1 The rainfall/runoff transformation model

The rainfall/runoff transformation model, DBS (Distributed Basin Simulator), is the core of RIBS. DBS models each grid cell in the distributed model as a two dimen-

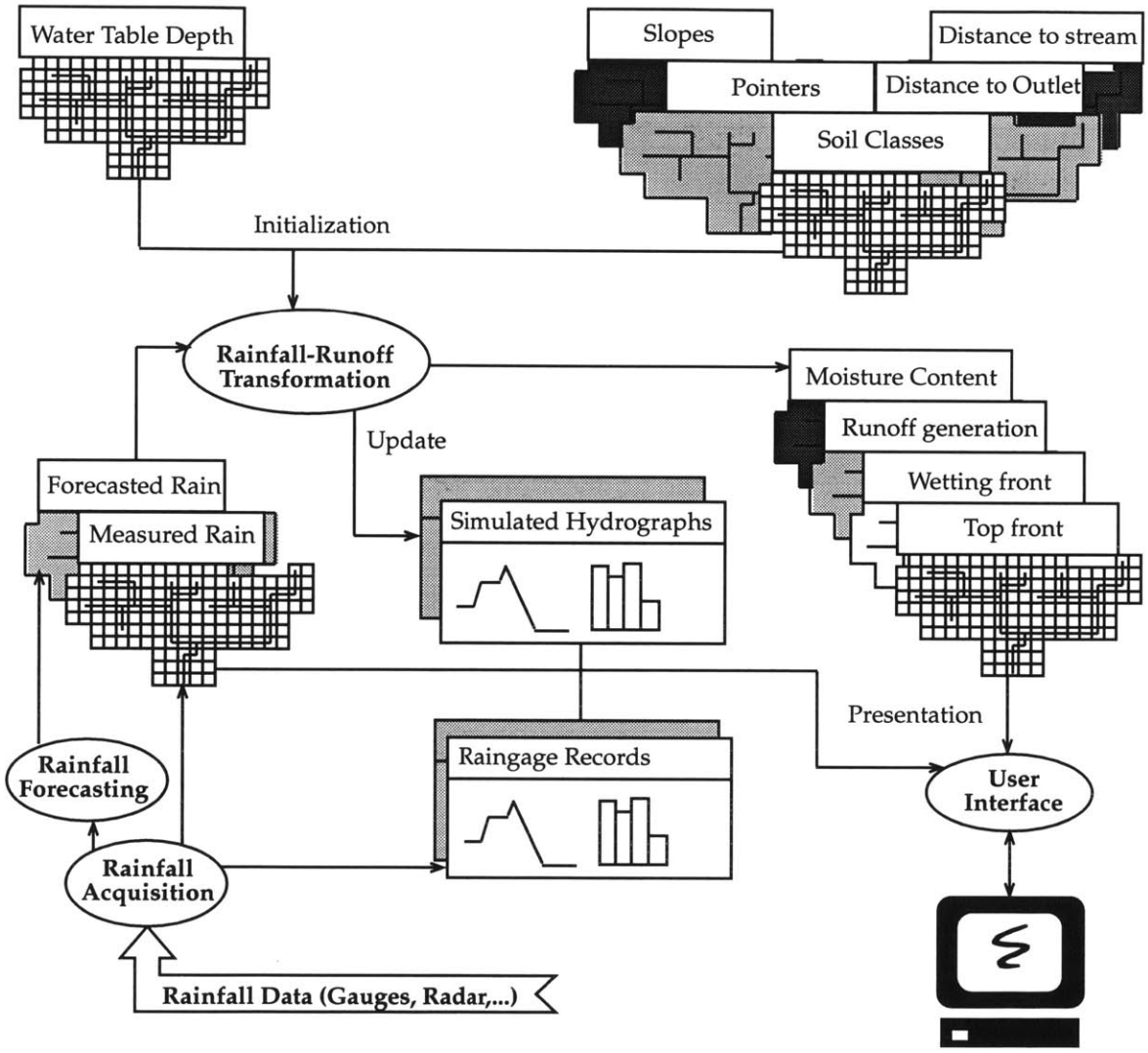


Figure 2-1: A general framework of the RIBS model (adapted from [26])

sional section of sloped soil (Figure 2-2). Within each grid cell, the internal dynamics of the cell are represented by a vertically layered, horizontally homogeneous soil system. The allowance for the hydraulic conductivity to decrease with depth in the soil column creates this vertical layering. Runoff is produced in DBS through two different methods. Runoff is generated when rainfall intensity exceeds the soil capacity at the surface (infiltration excess) and when the convergence of subsurface flow from upstream pixels and infiltration create a surface saturated area in the watershed (return flow). This runoff generation procedure is based upon a kinematic model of infiltration created by Cabral [12] which is detailed below.

The kinematic model of infiltration The model developed by Cabral is a one dimensional model of infiltration. It, nevertheless, allows for two dimensional flow in each grid cell due to anisotropy and the unique set of axes used in the model. The axes are defined perpendicular to the soil surface (n) and parallel to the sloping soil surface (p) as shown in Figure 2-2. The one dimensional infiltration process enters the pixel perpendicular to the ground surface along the n axis, but the flow of moisture within the pixel acts in both the n and p axes. Anisotropy is defined using an anisotropy ratio, a_r, which relates the saturated conductivities in the n and p directions as

$$a_r = \frac{K_{0p}}{K_{0n}} \tag{2.1}$$

where K_{0n} and K_{0p} are the saturated hydraulic conductivities at the surface. Hydraulic conductivity varies with depth in the following manner

$$K_{sn}(n) = K_{0n}e^{-fn} \tag{2.2}$$

$$K_{sp}(n) = K_{0p}e^{-fn} \tag{2.3}$$

where $K_{sn}(n)$ and $K_{sp}(n)$ are the saturated hydraulic conductivities at depth n from the surface of the sloped soil and f is an empirical parameter (L^{-1}) defining the rate of decay of conductivity in the soil.

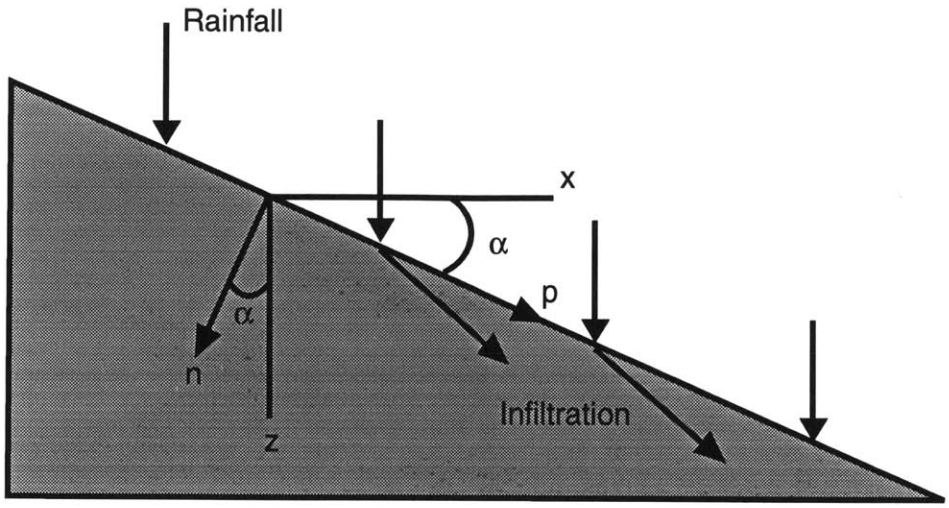


Figure 2-2: A two dimensional view of a pixel within RIBS

The dependence of hydraulic conductivity on soil moisture is given by the Brooks-Corey [10] parameterization. Combining the Brooks-Corey parameterization with equations 2.2 and 2.3, leads to the following expressions for unsaturated hydraulic conductivity

$$K_n(\theta, n) = K_{0n} e^{-fn} \left(\frac{\theta - \theta_r}{\theta_s - \theta_r} \right)^{\varepsilon} \tag{2.4}$$

$$K_p(\theta, n) = K_{0p} e^{-fn} \left(\frac{\theta - \theta_r}{\theta_s - \theta_r} \right)^{\varepsilon} \tag{2.5}$$

where $K_n(\theta, n)$ and $K_p(\theta, n)$ are the hydraulic conductivities in the n and p directions with a moisture content(θ) and a depth(n). Parameters defining saturated moisture content(θ_s), residual moisture content (θ_r), and pore size distribution index (ε) are also needed. These parameters could be defined as a function of depth, but the variability of these parameters with depth is quite small in comparison with other soil parameters and hence it is ignored ([14],[48]). Also, this approach has been used by others in unsaturated zone modeling with few problems ([18],[79]).

The kinematic approximation of flow in the unsaturated zone [6] implies gravity driven infiltration, with no capillary suction effects. Infiltrated moisture during a storm event is described by discrete fronts within the soil column. There can be

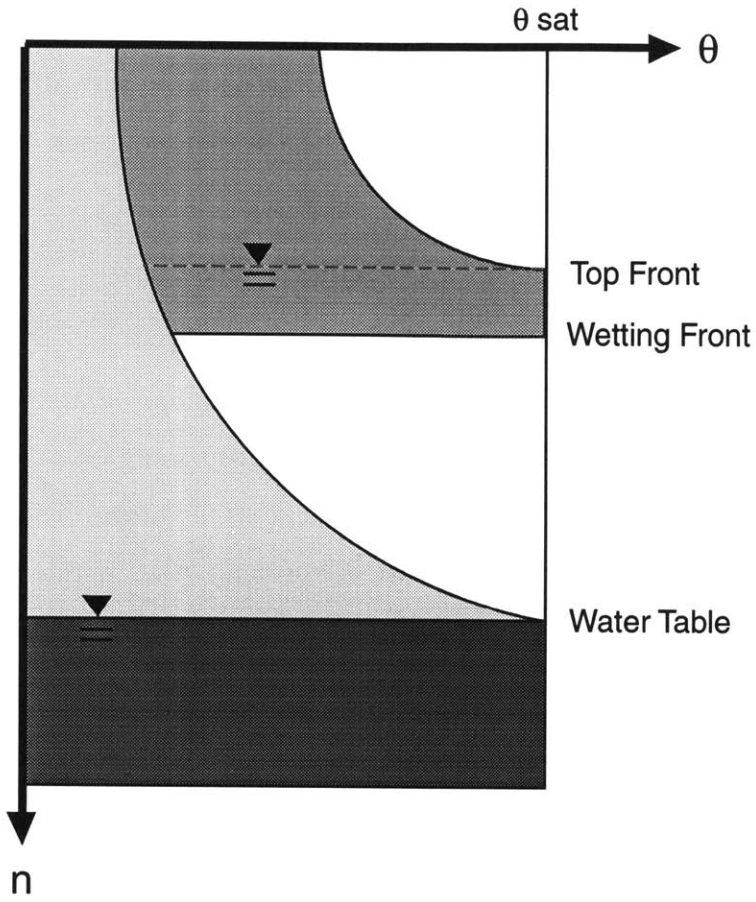


Figure 2-3: A representative pixel with wetting and top fronts

two fronts in the soil column, a wetting front and a top front. The wetting front represents the penetration depth of moisture during a storm event. The top front represents the height of a saturated zone which develops when the moisture flux into a pixel is greater than the hydraulic conductivity of the soil at the present depth of the wetting front. Figure 2-3 shows an example of a typical grid cell soil column with a wetting and top front.

Assuming constant normal flux of R, the infiltration rate, in the unsaturated zone and using the kinematic approximation leads to $R = K(\theta, n)$. Using this and the Brooks-Corey equations given earlier (equations 2.4 and 2.5) one can solve for soil moisture, θ, as follows

$$\theta(R,n) = \left(\frac{R}{K_{0n}}\right)^{\frac{1}{\epsilon}} (\theta_s - \theta_r) e^{\frac{f}{\epsilon} n} + \theta_r \tag{2.6}$$

For every infiltration rate R, there is a corresponding soil column depth (designated N^*), where the saturated hydraulic conductivity will be equal to the infiltration rate. One can solve to find this depth, N^*, by setting $K_{sn} = R$ in equation 2.2 and rearranging to obtain

$$N^*(R) = \frac{1}{f} ln \left(\frac{K_{0n}}{R} \right) \tag{2.7}$$

This depth, N^*, represents the depth at which saturation develops. Water can no longer flow downward fast enough to fully transmit R, and a top front with perched saturation is created within the soil column to handle this excess moisture. The above derivation for N^* only applies when the rainfall rate is less than K_{0n}. If the rainfall rate is greater than K_{0n}, the entire soil column will be saturated from the surface to the wetting front.

The movement of the wetting and top fronts depends on moisture fluxes at each time step. Typical front movement can be shown by looking at a wetting cycle in DBS. In a wetting cycle (Figure 2-4), a small wedge of moisture first appears in the soil column (panel 1). As the rain continues, this wedge will move down and saturation will occur at the point where moisture flux into the pixel becomes greater than the hydraulic conductivity at N^* (panel 2). At this point a top front will be created and it will move towards the soil surface. When the top front reaches the soil surface (panel 3), the pixel will be surface saturated, creating runoff. Further rain keeps the surface saturated while the wetting front moves to the water table (panel 4). The equations that detail these movements are quite complex, and differ based upon the moisture content of the pixel.

If the wetting front has not reached saturation and the rainfall rate is less than K_{0n} (Figure 2-4, panel 1), both the wetting and top fronts are at the same location in the soil column. The time evolution of the wetting front is derived by integrating the continuity equation. In the n and p coordinate system, the continuity equation is given by

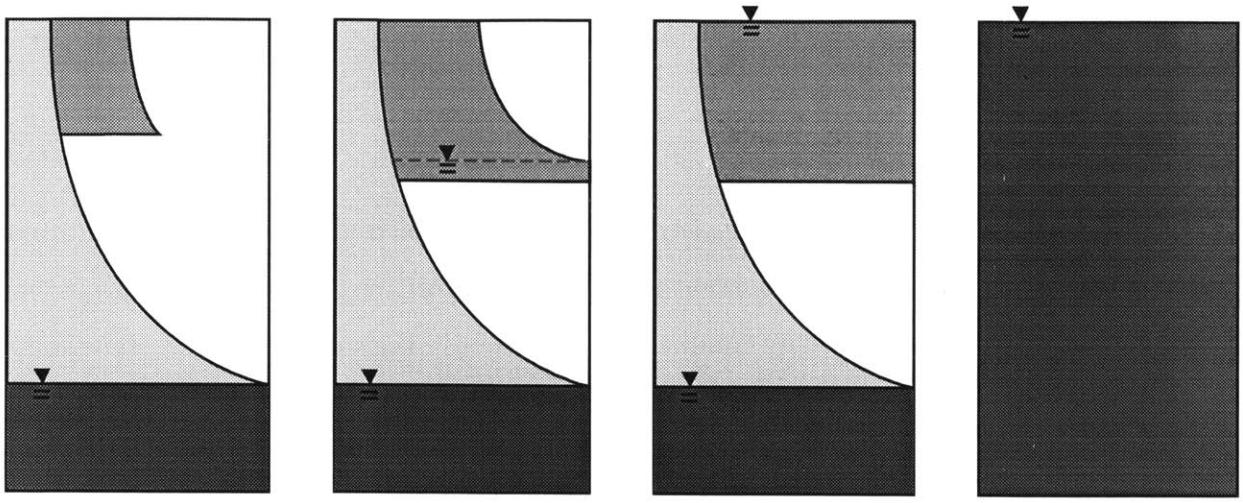


Figure 2-4: An example of a wetting cycle within RIBS

$$\frac{\partial \theta}{\partial t} + \frac{\partial q_n}{\partial n} + \frac{\partial q_p}{\partial p} = 0 \tag{2.8}$$

where q_n and q_p are the discharges per unit area in the n and p directions. After integration over the entire wetting front, the following expression for the evolution of the wetting front is obtained

$$\frac{dN_f}{dt} = \frac{(R - R_i)cos(\alpha)}{\theta(R, N_f) - \theta(R_i, N_f)} \tag{2.9}$$

where N_f is the depth of the wetting front, α is the angle of the pixel, R is the infiltration rate, and R_i is the initial recharge rate defined at the beginning of the storm event. (see [12] for the full derivation)

When the wetting front reaches saturation (Figure 2-4, panel 2), the top front and wetting front are not equal. When N_f reaches N^*, a top front is created which moves upwards through the soil column. At the same time, the wetting front continues to descend. The equations defining the evolution of wetting and top front in this case are similar in structure to the equation given in the unsaturated zone (equation 2.9). The major change involves the use of q_n instead of infiltration rate, R, in the formulation.

To find q_n in the perched saturated area of a pixel, begin with the continuity equation (2.8). Under saturation with derivatives in the p direction assumed to be

zero, the continuity equation reduces to

$$\frac{\partial q_n}{\partial n} = 0 \tag{2.10}$$

The above result states that $q_n(n)$ is a constant in the n direction under saturated conditions. To support this constant normal flow, a positive pressure will build in the soil column due to the varying conductivities present in the saturated zone. In assuming that the pressure at fronts N_f and N_t are atmospheric, a pressure distribution, $\Psi(n)$, can be determined for the saturated zone as follows

$$\Psi(n) = cos(\alpha)\left[\left(n+\frac{1}{f}\right) - \frac{e^{fN_f} - e^{fn}}{e^{fN_f} - e^{fN_t}}\left(N_t + \frac{1}{f}\right) - \frac{e^{fn} - e^{fN_t}}{e^{fN_f} - e^{fN_t}}\left(N_f + \frac{1}{f}\right)\right] \tag{2.11}$$

When differentiating this pressure distribution, the hydraulic potential is obtained. Normal flow in the saturated area can be found by substituting the normal component of the hydraulic potential in the flow equation. This results in the equation below

$$q_n = K_{0n}\frac{f(N_f - N_t)}{e^{fN_f} - e^{fN_t}}cos(\alpha) \tag{2.12}$$

With a calculated q_n, top front and wetting front evolution in the saturated zone can be computed as follows

$$\frac{dN_f}{dt} = \frac{(q_n - R_i cos(\alpha)}{\theta_s - \theta(R_i, N_f)} \tag{2.13}$$

$$\frac{dN_t}{dt} = \frac{(q_n - R cos(\alpha)}{\theta_s - \theta(R, N_t)} \tag{2.14}$$

(See Cabral [12] for the full derivation)

Modifications to the Infiltration Model This kinematic model of infiltration operates on a single pixel under constant rainfall intensity. This condition is significantly different than that needed in a basin scale model of flood forecasting. Therefore, several significant assumptions and modifications were added to incorporate flow

aggregation, varying front positions, and spatially and temporally varying rainfall.

The variability in rainfall is accounted for by assuming that the redistribution of water is almost instantaneous within the moisture wedge created by the model. This allows for continued use of uniform normal flow in the pixel with a single moisture front. The redistribution of moisture is obtained by defining an equivalent uniform recharge/infiltration rate, R_e, that leads to a moisture profile that when integrated over the unsaturated zone equals the moisture content, M_u, in that zone. Integrating the soil moisture profile (equation 2.6), over the entire unsaturated area and equating it to M_u gives

$$\int_0^{N_t} \left[\left(\frac{R_e}{K_{0n}} \right)^{\frac{1}{\varepsilon}} (\theta_s - \theta_r) e^{\frac{f}{\varepsilon} n} + \theta_r \right] dn = M_u \tag{2.15}$$

solving and rearranging for R_e leads to

$$R_e = K_{0n} \left(\frac{M_u - \theta_r N_t}{(\theta_s - \theta_r) \frac{\varepsilon}{f} (e^{\frac{f N_t}{\varepsilon}} - 1)} \right)^{\varepsilon} \tag{2.16}$$

With a known R_e, the equations for front evolution (equations 2.9, 2.13, and 2.14), can be used effectively under varying rainfall with the simple replacement of R with R_e.

As mentioned earlier, the kinematic flow model on a sloping pixel produces lateral flow due to geometry, anisotropy, and heterogeneity in the soil during the infiltration process. Lateral flow is also produced due to pressure and moisture gradients between adjacent pixels. The flows produced due to the infiltration process are determined by integrating pressure gradients along a vertical surface of the pixel and obtaining flow across that vertical surface. This integration process leads to

$$Q = W sin(\alpha) \left\{ [N_t R(a_r - 1)] + \left[K_{0n} \frac{a_r}{f} (e^{-f N_t} - e^{-f N f}) \right] - \left[K_{0n} \frac{f(N_f - N_t)^2}{e^{f N_f} - e^{f N_t}} \right] \right\} \tag{2.17}$$

where W is the width of the vertical cross section of the pixel. By applying this recursively beginning with the headwaters, the outflows from upstream pixels become

the inflow to downstream elements, conserving water in the system.

The lateral flow resulting from variability between adjacent pixels has been shown to be fairly small with the respect to lateral flows due to topography and anisotropy ([21],[28]). Therefore, a simplified procedure is used to determine these fluxes, using lateral pressure imbalances to calculate this moisture transfer. A Darcian flow equation used to solve for this lateral flow is given by

$$q_x(z) = -K_{eq}(z)J_x(z) \tag{2.18}$$

where $q_x(z)$ is the lateral flow in the pixel, $K_{eq}(z)$ is an equivalent conductivity, and $J_x(z)$ is the lateral gradient. For this Darcian calculation, a uniform slope is calculated by connecting the centers of each pixel representing an average slope across two adjacent pixels. With this simplification, the lateral gradient can be defined by

$$J_x(z) = \frac{\partial \Psi}{\partial x} \approx \frac{\Delta \Psi(z)}{\Delta x} = \frac{\Psi_2(z) - \Psi_1(z)}{x_1 - x_2} \tag{2.19}$$

where $x_2 - x_1$ is the lateral distance between pixel centers. The equivalent hydraulic conductivity, K_{eq} is needed to find the lateral flow and is calculated using a combination of the two pixel conductivities.

$$K_{eq} = \sqrt{K_{0p1}K_{0p2}}\, e^{-fzcos(\alpha)} \tag{2.20}$$

Combining equations 2.18, 2.19, along with the expression for Ψ derived in equation 2.11, an integration over the entire saturated area can be completed to define the lateral flow, *Qpout*

$$Q_{pout} = K_{eq}\frac{W}{x_2 - x_1}(N_f - N_t)\left[\frac{N_t e^{fN_f} - N_f e^{fN_t}}{e^{fN_f} - e^{fN_t}} + \frac{1}{f} - \frac{N_f + N_t}{4}\right] \tag{2.21}$$

Runoff Generation Both infiltration excess runoff and return flow are represented in the DBS model. Using the details of moisture front location (N_t and N_f) computed earlier, a series of pixel states are defined for these front positions. With a defined

pixel state, the infiltration excess runoff is computed. After calculating infiltration excess runoff, the return flow is determined by analyzing the moisture balance in each pixel due to the subsurface inflows and outflows calculated with the lateral moisture transfer equations (2.17 and 2.21).

Four basic pixel states are considered within the DBS model, each defined by a different set of moisture front positions within the soil column. These four states have different runoff potentials, and are defined as unsaturated, perched saturated, surface saturated, and fully saturated pixel states (Figure 2-5). The unsaturated pixel contains a wetting front (N_f) which has not yet reached saturation within the soil column. Only infiltration excess runoff will occur in this pixel state. Similarly, the perched saturated pixel state will also only produce runoff under infiltration excess. This state differs from the unsaturated state in that the wetting front has reached saturation, and a top front has developed. When the wetting front is some depth above the water table, but the top front is at the surface, a surface saturated pixel state is created. At this point, infiltration excess runoff increases drastically, as only some of the incoming water will be used to move the wetting front downward. All other moisture will generate runoff. When the wetting front reaches the water table, the pixel is fully saturated. All moisture input to the pixel will immediately generate infiltration excess runoff.

The actual amount of infiltration excess runoff will be based upon the maximum infiltration capacity, I_{max}, of the given pixel. For unsaturated and perched saturated pixels, this capacity is only limited by the surface saturated hydraulic conductivity

$$I_{max} = K_{0n} cos(\alpha) \tag{2.22}$$

As described earlier, when the pixel is in the surface saturated state, the infiltration capacity is drastically diminished as moisture capacity is limited by the slow downward movement of the wetting front. This results in an infiltration capacity of

$$I_{max} = K_{0n} \frac{f N_f}{e^{f n_f} - 1} cos(\alpha) \tag{2.23}$$

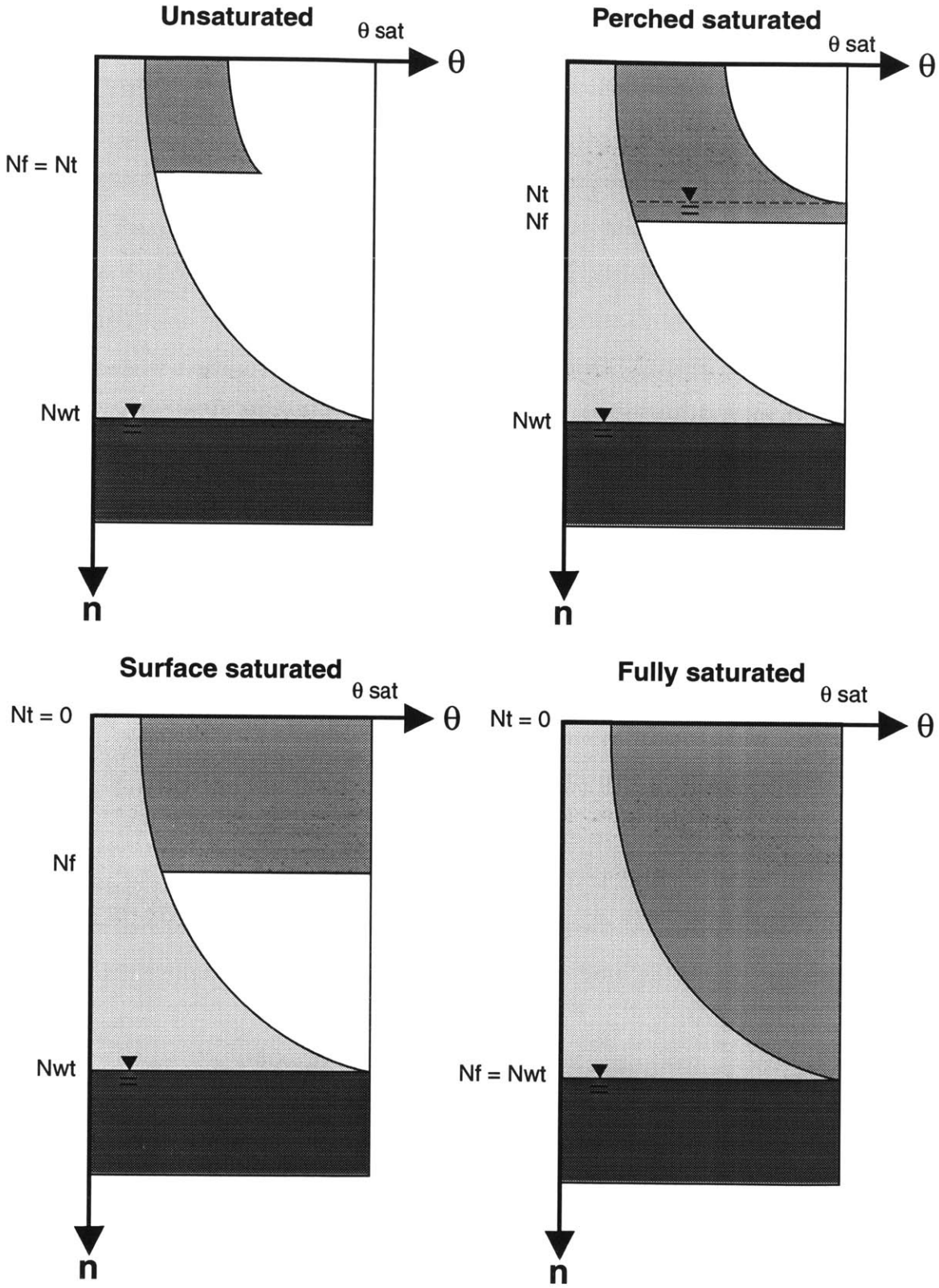


Figure 2-5: The four possible pixel states allowable in the RIBS model

In the fully saturated pixel state, no water can infiltrate the soil leading to

$$I_{max} = 0 \tag{2.24}$$

For a rainfall rate, R, the infiltration excess runoff, R_{infil}, can be defined as

$$R_{infil} = R - I \tag{2.25}$$

where I, the actual infiltration, is given by

$$I = R \quad \text{if} \quad R \leq I_{max} \tag{2.26}$$

$$I = I_{max} \quad \text{if} \quad R > I_{max} \tag{2.27}$$

Return flow only occurs when a pixel is saturated. Therefore, only under the surface saturated and fully saturated conditions will this process be considered. All moisture inflows in the model are assumed to accumulate in the area above the wetting front. With this assumption, the maximum total moisture content in a pixel is always limited to $N_f\theta_s$. If due to inflows, the moisture content exceeds this limit, return flow will be produced. In a surface saturated pixel, the amount of return flow produced is governed by the speed of descent of the wetting front. Mathematically, this is defined as

$$I + \frac{Q_{pout} - Q_{pin}}{A} > \frac{dN_f}{dt}[\theta_s - \theta(R_i, N_f)] \tag{2.28}$$

where A is the horizontal area of the element. The first term in equation 2.28 details the infiltration process, the second term defines the lateral inflow, and the final term represents the rate of descent of the wetting front. In this condition, return flow, R_r, will be equal to

$$R_r = I + \frac{Q_{pout} - Q_{pin}}{A} - \frac{dN_f}{dt}[\theta_s - \theta(R_i, N_f)] \tag{2.29}$$

Under fully saturated conditions, $\frac{dN_f}{dt} = 0$ and R_r is at a maximum of

$$R_r = I + \frac{Q_{pout} - Q_{pin}}{A} \tag{2.30}$$

The total runoff from each pixel, R_f, is then determined by adding the infiltration excess runoff calculated in equation 2.25 and the return flow calculated above (equation 2.30). This total runoff from each pixel is then routed to the outlet using the method given in the next section.

2.1.2 Surface Flow Routing

After computing the runoff from each pixel, this flow must be routed to the outlet. To find the response of a watershed, DBS uses the distributed convolution equation

$$Q(t) = \int_a \int_0^t R_f(x, y, \tau) h(x, y, t - \tau) d\tau dA \tag{2.31}$$

where Q(t) is the hydrograph at the outlet, $R_f(x, y, \tau)$ describes the runoff generated per unit area, and h(x,y,t) is the instantaneous response of an element dA located at (x,y). This instantaneous response function, h(x,y,t), is approximated by neglecting dispersion in the response function. A Dirac delta function is then used to define h(x,y,t) with a delay equal to the travel time as follows

$$h_\tau(x, y, t) = \delta \left(\frac{l_h(x, y)}{\nu_h(\tau)} + \frac{l_s(x, y)}{\nu_s(\tau)} \right) \tag{2.32}$$

where $l_h(x, y)$ and $l_s(x, y)$ are the hillslope and stream channel lengths in the path from pixel (x,y) to the basin outlet, and $\nu_h(\tau)$ and $\nu_s(\tau)$ are the hillslope and channel velocities at time τ. These travel distances can be found quite easily in a standard Digital Elevation Model and are determined using routines developed by Tarboton [73].

For every time step, τ, a basin response routing the runoff generated at every pixel is created by

$$q_\tau(t) = \sum_{(x,y)\in Basin} R_{f\tau}(x,y)h_\tau(x,y,t)\Delta x\Delta y \tag{2.33}$$

where $R_{f\tau}$ is the runoff generation form each pixel as determined by equations 2.25 and 2.30 and $\Delta x\Delta y$ is the area of the pixel. The total basin response for a storm length T is then determined by summing the response at each time step

$$Q(t) = \sum_{\tau=0}^{\tau=T} q_\tau(t) \tag{2.34}$$

The previous system of equations for surface routing describe a linear system. To add non-linear basin response to the DBS model, a power law is defined to describe the flow velocity in the watershed

$$\nu_s(\tau) = C_\nu[Q(\tau)]^r \tag{2.35}$$

where ν_s is the stream channel velocity, $Q(t)$ is the discharge at the outlet at time step τ, and C_ν and r are coefficients estimated through calibration. The associated hillslope velocity is given by

$$\nu_h(\tau) = \frac{\nu_s(\tau)}{K_\nu} \tag{2.36}$$

where K_ν is a constant ratio of stream velocity to hillslope velocity found through model calibration.

2.2 CHILD

Created as a landscape evolution model, the CHILD (Channel-Hillslope Integrated Landscape Development) model has been used in this project is due to its unique model framework. While modeling drainage basin evolution is the main application of the model, CHILD can also be looked at as a software development tool. Written in C++, CHILD provides a data structure and model algorithms which can be reused to create the new version of the RIBS model in a TIN format. CHILD provides a

general framework for storing and accessing the triangular mesh along with providing methods for computing mass fluxes and creating drainage paths across the TIN [74].

2.2.1 Model Overview

The CHILD model is a collection of geomorphic process modules. These process modules include a storm generation, runoff generation mechanisms, water and sediment routing, erosion, sediment transport, meandering, floodplain deposition, and tectonic deformation [75]. A schematic summarizing all of the processes in CHILD is shown in Figure 2-6. Due to the modular structure of the model, these modules can be added or removed to a particular modeling exercise as necessary. CHILD also includes routines and data structures to manage the spatial and temporal needs of the process modules. The spatial framework includes an adaptive, irregular, mesh which allows for varying spatial resolution within the model. This unique spatial framework will be discussed in detail below.

2.2.2 Model Framework

The irregular mesh within CHILD is created by a set of points in any user specified configuration. These irregularly spaced points are then connected using a Delaunay triangulation which minimizes the maximum interior angles created by a set of irregular points. The Delaunay triangulation creates the most "regular" triangles, avoiding the "fat" triangles created by other triangulation schemes [77]. From these Delaunay triangles, one can create a system of voronoi (Thiessen) polygons. These polygons are created by connecting the perpendicular bisectors of the triangle edges (see Figure 2-7). Well established computational methods to create the Delaunay triangles and the associated voronoi polygons already exist and have been well tested ([71],[43],[62], and many others).

The data structure used to store the irregular mesh data is very similar to that used by Braun and Sambridge in 1997 [9]. Modified for use in CHILD [74], the data structure was designed for finite-difference modeling. Therefore, state variables are

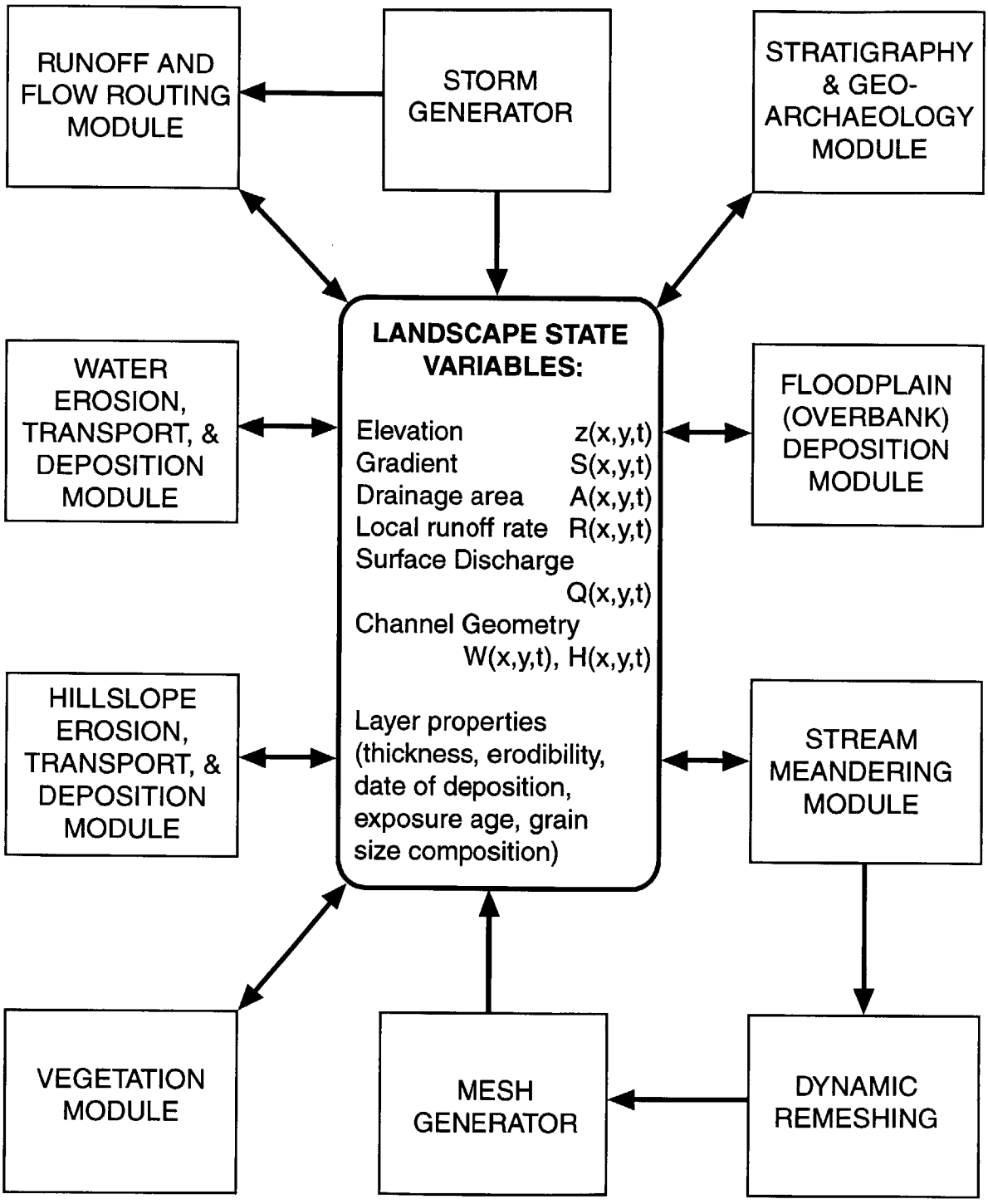


Figure 2-6: A schematic look at the CHILD model processes. (Adapted from [75])

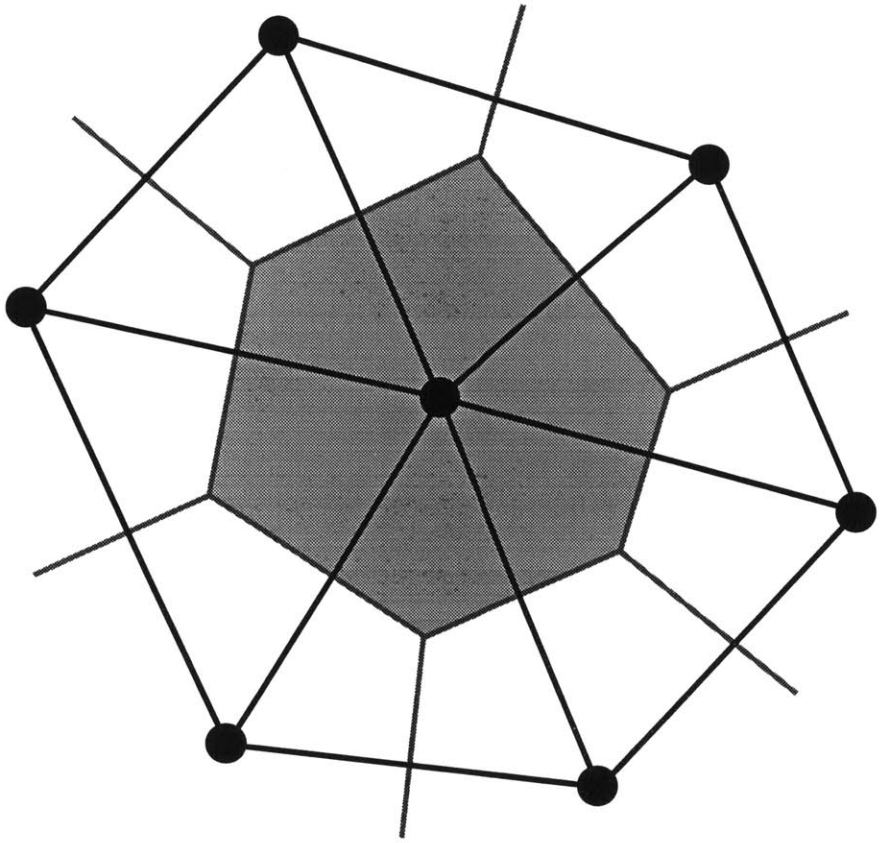


Figure 2-7: A simple TIN with an associated voronoi area. The voronoi area for the node at the center of the figure is shaded in gray)

defined for the nodes (points) instead of the triangles. Each of these nodes also has an associated voronoi area (Figure 2-7). The data structure for this triangular mesh was designed to allow for quick retrieval of adjacent mesh elements while keeping storage requirements low. The data structure also had to have the flexibility to handle dynamic changes in the mesh, adding and deleting nodes while the model is running. With these requirements, a "dual edge" structure was used to store the mesh [74]. This "dual edge" structure was derived from the Quad Tree structure of Guibas and Stolfi [32], and describes the mesh as three major data elements: nodes, triangles, and directed edges.

Node Structure Each node in the system includes the following: a set of x,y, and z coordinates, the number of neighbor nodes, and a reference (pointer) to a directed edge which originates at that node. Only one reference to a directed edge is needed as all other edges that originate at that node can be found through pointers defined for that edge. Other attributes can also be included as part of the node object. In the case of CHILD, the number of edges associated with each node and a boundary code defining the status of the node (interior and exterior point) are included. Using Figure 2-8 as an example, the model data structure would create a node object which resembles Table 2.1. Due to the flexible C++ coding of this data structure, this object can also be encapsulated as part of a more complex node structure including other information for the modeling exercise (for example, voronoi area and node parameters). These encapsulation methods will be shown in detail when modifying the CHILD model in Chapter 3.

Directed Edges As part of the complete data structure, the data representing the edges of the triangles must also be stored. A directed edge connects two nodes and has a origin and destination node giving it a direction. With a directed edge, each edge of a triangle is actually represented twice. As an example, the edge between nodes A and B in Figure 2-8 is given by two directed edges, edge a and edge b. These are different directed edges because edge a points from node A to node B while edge

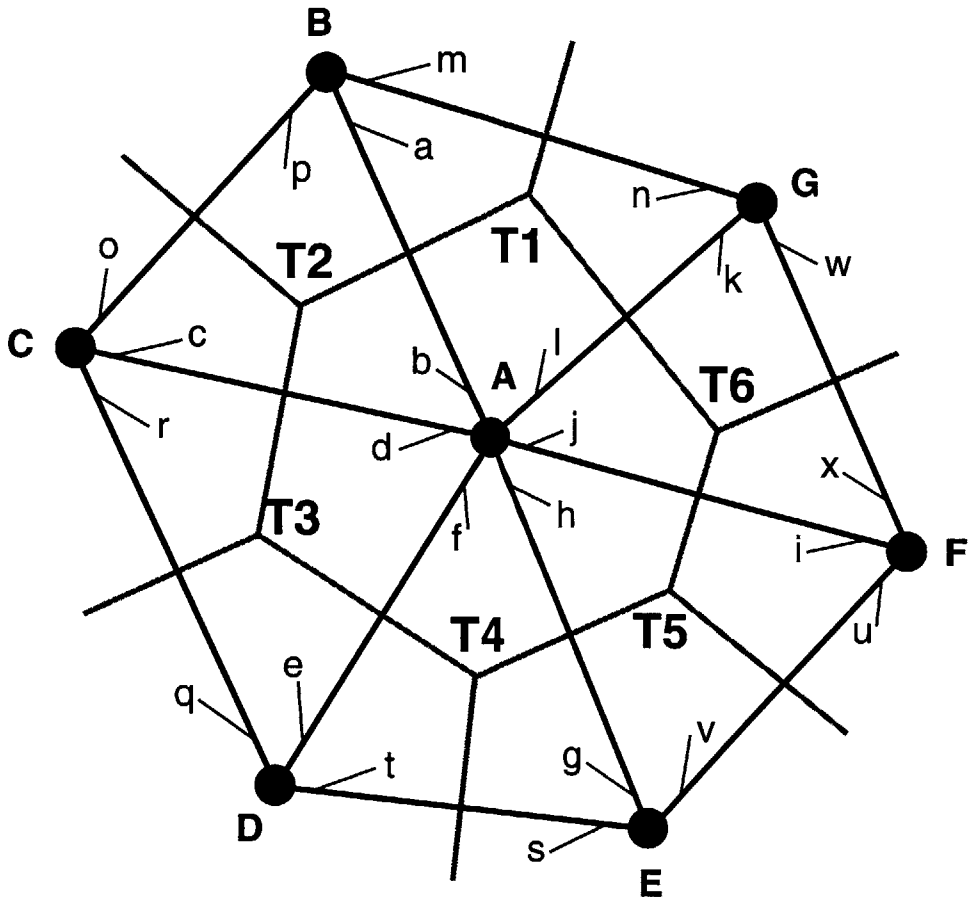


Figure 2-8: An example of a TIN and the associated voronoi areas. Nodes are labeled with capital letters, small letters define directed edges and all labels with a 'T' prefix define the triangles. Adapted from [74]

Table 2.1: The node structure for Figure 2-8

Node	x	y	z	EDG	#NBRS	Boundary Code
A	Xa	Ya	Za	a	6	0
B	Xb	Yb	Zb	o	3	1
C	Xc	Yc	Zc	q	3	1
D	Xd	Yd	Zd	r	3	1
E	Xe	Ye	Ze	h	3	1
F	Xf	Yf	Zf	w	3	1
G	Xg	Yg	Zg	m	3	1

Table 2.2: The directed edge structure for Figure 2-8

Directed Edge	Origin Node	Dest. Node	CCW Edge	RH Voronoi Vertex
a	A	B	c	CC(T1)
b	B	A	n	CC(T2)
c	A	C	e	CC(T2)
d	C	A	p	CC(T3)
e	A	D	g	CC(T3)
f	D	A	r	CC(T4)
etc...				
m	G	B	l	(NULL)
n	B	G	o	CC(T1)
etc...				

b points from node B to node A. In the model structure, each directed edge contains pointers to the origin node, destination node, and the directed edge located counter-clockwise to the origin node. Coordinates of the voronoi vertex located to the right of the directed edge are also included in this structure. An example list of the edges produced from Figure 2-8 is given as Table 2.2. Like the node structure, the edge data object can be encapsulated to handle other variables as needed for specific modeling exercises.

Triangle Data Structure The third data object needed to complete the mesh is a triangle data object. Included in these triangle data objects are pointers to three nodes in the triangle, three neighboring triangles, and three directed edges that make up the triangle. The edges are chosen using the three edges oriented counter clockwise within the triangle. Table 2.3 presents an example triangle data structure derived from Figure 2-8. Additional data can be included in triangle data structure as needed.

2.2.3 Model Algorithms

Flow Routing on a TIN Modeling flow on a TIN has normally been accomplished using a "triangle-based" approach ([57],[25],[41],[51]). These models define flow across

Table 2.3: The triangle data structure for Figure 2-8

Triangle	Nodes	Adjacent Triangles	CCW-oriented edges
T1	B, A, G	T6, -1, T2	n, a, l
T2	A, B, C	-1, T3, T1	c, b, p
T3	C, D, A	T4, T2, -1	d, r, e
T4	D, E, A	T5, T3, -1	f, t, g
T5	E, F, A	T6, T4, -1	h, v, i
T6	F, G, A	T1, T5, -1	j, x, k

the triangle surfaces along with flow on the triangle edges, using interpolation schemes to find the gradient of flow in each triangular surface (see Figure 2-9). This creates a precise definition of surface flow in the TIN where flow is not constrained to any predefined pathways. The major drawback of this method is that it is computationally intense. The two different flow systems (flow across triangles and along edges) must be computed separately which adds significant complexity to the modeling system. Due to this major drawback, a "triangle-based" approach is not used in the CHILD model.

As an alternative, a "voronoi-based" approach to flow routing on a TIN has been also been developed [9]. With slight modifications, this is the flow routing scheme in CHILD. In this approach, each voronoi area in the watershed acts like a grid cell in a traditional raster model. All flow within the voronoi cell is concentrated at the node located in the center of the voronoi area. This flow is then routed to a downhill node along the steepest edge connected to the original node. This modeling approach does limit flow pathways in the watershed as flow is constrained to the edges of the constructed triangles.

The similarity to traditional grid-based modeling allows for simple computation of other hydrologic algorithms and processes. Raster-based methods for filling pits in a watershed and determining contributed area can be used with slight modifications. Ordering the nodes in a watershed can be completed in a "voronoi-based" system. Sorting the nodes from headwater (upstream) to outlet (downstream) is computed

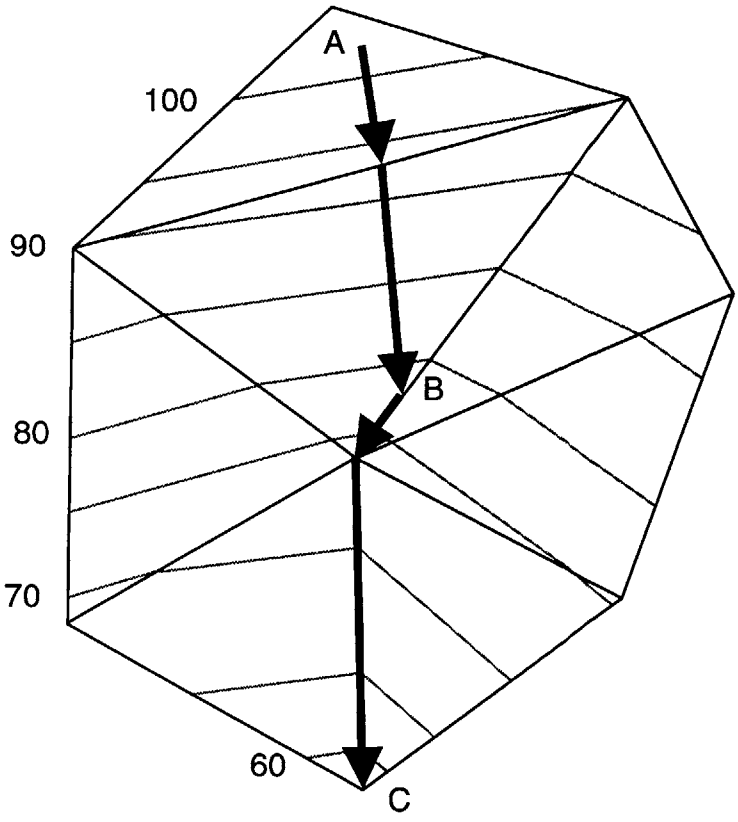


Figure 2-9: Surface flow in a traditional TIN based model: If a raindrop hits the surface at point A, it will flow across the triangle face perpendicular to the contour lines (in gray) until point B. At this time, flow continues along the triangle edges to the basin outlet at point C.

using a cascade algorithm from Braun and Sambridge [9].

One and Two Dimensional Transport in a TIN In using the "voronoi-based" approach, each voronoi polygon can be modeled as a finite-difference cell. With this system, continuity of mass can be written as

$$\frac{dV_i}{dt} = \sum_{j=1}^{N_i} Q_{ji} \tag{2.37}$$

where V_i is the volume stored in the node i, N_i equals the number of nodes connected to node i, and Q_{ji} represents the flux from node j to node i.

In the CHILD model, one dimensional transport of sediment and water is calculated using a cascade of one dimensional objects. The model calculates sediment and water transport at each node beginning at the headwater moving downstream to the outlet at the final step. As a representative one dimensional process, fluvial erosion or deposition are modeled as using

$$\frac{dz_i}{dt} = \frac{\left(\sum_{j=1}^{n} Q_{sj}\right) - Q_{si}}{(1-\nu)\Lambda_i} \tag{2.38}$$

where z_i is the elevation of node i, n is the number of nodes flowing to directly to node i, Q_s is sediment flux, ν represents porosity, and Λ_i is equal to the voronoi area of node i ([9],[74]). Solved using forward difference, matrix, or other numerical methods, this one dimensional transport approach will also be used in distributed hydrologic modeling (see Chapter 3).

Diffusive transport processes in two dimensions require a different method. To compute mass exchange in the finite difference mesh, the interface width must be known. In the TIN system presented, the width of the shared voronoi cell edge (Figure 2-10) will be used to approximate this width [9]. In CHILD, this two dimensional diffusive transport method is used to solve for sediment transport. As a linear function of surface gradient [17], sediment transport per unit width, q_s, is given by

$$q_s = k_d \frac{\partial z}{\partial x} \tag{2.39}$$

where k_d is the diffusivity constant, and x is a vector in the downslope direction [74]. Combining the mass continuity equation (2.37) with the sediment transport relation (2.39), an approximate numerical solution calculating the rate of change in elevation at each node due to diffusion can be given by

$$\frac{dz_i}{dt} = -\frac{k_d}{\Lambda_i} \sum_{j=1}^{n} \lambda_{ij} \frac{(z_i - z_j)}{L_{ij}} \tag{2.40}$$

where n is the number of nodes directly adjacent to node i, L_{ij} is the length of triangle edge connecting nodes i and j, and λ_{ij} is the width of the shared voronoi edge [74]. This type of diffusion process also describes groundwater flow quite well. This groundwater process will be used in the distributed hydrologic model and is described in Chapter 3.

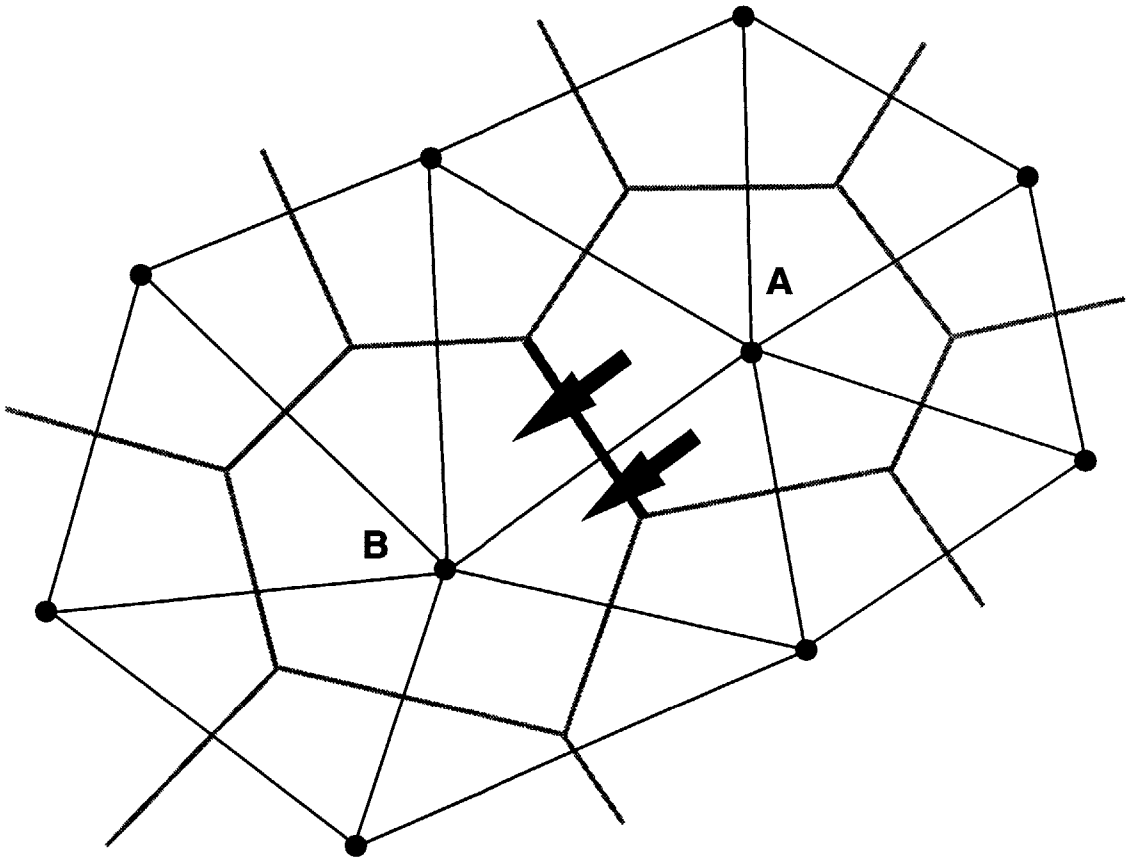


Figure 2-10: Two Voronoi cells and their shared voronoi edge. The shared voronoi edge for nodes A and B is highlighted in red.

2.3 The Sacramento Model

The Sacramento Soil Moisture Accounting (SAC-SMA) Model is a spatially lumped, conceptual, model of the land phase of the hydrologic cycle [11]. The model receives rainfall, evaporation, snow cover, and snow melt as input and produces a time series of channel flow as output. The model is typically run at a 6 hour time step and requires approximately eight years of historical data for proper calibration. This model is used extensively by the National Weather Service to forecast floods and it used primarily on basins from 300 to 5000 km^2 [68].

The SAC-SMA Model consists of a series of boxes designed to represent physical processes in rainfall/runoff modeling. These boxes work as buckets in the model, storing water until they are filled. As the buckets overflow, runoff is generated in the model. A schematic of all the buckets in the SAC-SMA model is given as Figure 2-11. The pervious zone is represented by two soil layers, an upper zone and a lower zone. The major source of storm runoff, the upper zone is an active permeable zone near the surface of the catchment. The lower zone is the deep soil layer and baseflow is generated here. Both of these zones contain free and tension water components. The tension water is held tightly by the soil and can only be removed through evaporation. The free water can move vertically (percolation) and horizontally (interflow and baseflow) through the soil. An impervious portion is also modeled in the SAC-SMA generating impervious and direct runoff.

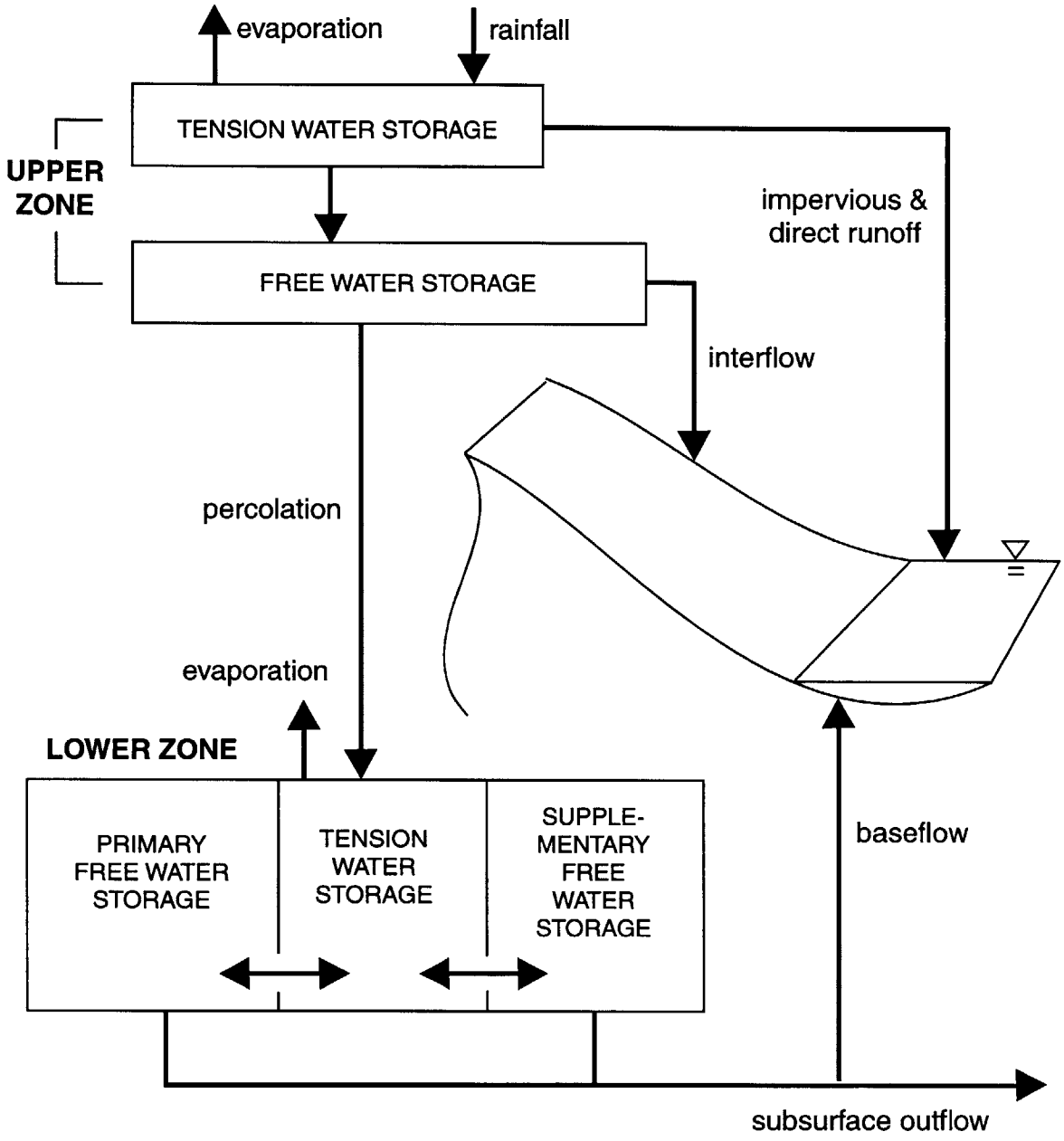


Figure 2-11: A schematic diagram of the Sacramento Model

Chapter 3

Moving from RIBS to tRIBS

In creating the tRIBS model, the RIBS and CHILD models were modified and integrated into one coherent system. This process was accomplished by first modifying the CHILD model. The CHILD model provides the framework of the system, creating the simulation structure. This includes model input/output, the TIN mesh, and simulation timer. The RIBS model provides the specifics that pertain to flood forecasting in the system. This includes runoff and unsaturated and saturated groundwater dynamics at each pixel. The paragraphs below describe the methods used to accomplish this task.

3.1 Modifications to the CHILD model

To move RIBS into the CHILD model framework, numerous changes were made to the CHILD system. The first major change involved the addition of new variables to the node data structure of CHILD. As described in Chapter 2, the CHILD model allows for the templating of the node structure. This templating provides a mechanism for adding variables to the CHILD model without creating errors in earlier versions of the system. The complete list of variables from RIBS that were included as part of the node structure of CHILD is given in Table 3.1. This new node data structure was also modified to use the concept of data 'hiding' available in an object oriented programming language. This data hiding provides an extra layer of security in tRIBS

Table 3.1: Variable included in the tRIBS node template

Variable Name	Description
Nwt	Water table depth
Mu	Moisture content in the unsaturated zone
Mi	Initial moisture content
Nf	Wetting front depth
Nt	Top front depth
Ri	Initial unsaturated zone recharge
Ru	Recharge in the unsaturated zone
alpha	Pixel slope
QpOut	Lateral Flow out of a pixel
QpIn	Lateral Flow into a pixel
srf	Total runoff generated
hsrf	Hortonian runoff
esrf	Exfiltrated runoff
sbsrf	Saturated from below runoff
psrf	Perched saturation runoff
flowedge	Edge that the runoff flows through
traveltime	Travel time from pixel to outlet
intstorm	Time since the last storm

as data cannot be accessed directly and can only be changed with the use of member functions of the node object. The complete code for this modified node structure is given as tCNode.cpp and tCNode.h in Appendix B.

The original CHILD model was not designed to incorporate inputs after the beginning of the simulation. For tRIBS this was changed as time varying rainfall is a necessary input to a flood/forecasting model. A flexible new object, tRainfall, was created for this purpose. Based upon the number of voronoi cells in the system, tRainfall creates a dynamic array to hold the rainfall for each time step. This object uses the rainfall file created in Chapter 4 as its input. The rainfall file is not ordered by node ID. Therefore, the tRainfall object was written to capture both the ID and rainfall amount from the input data file. To allow for real-time use of the tRIBS model, rainfall is input to the model as the simulation in running. At each time step, the rainfall for that particular time is read into the model. This overwrites the previ-

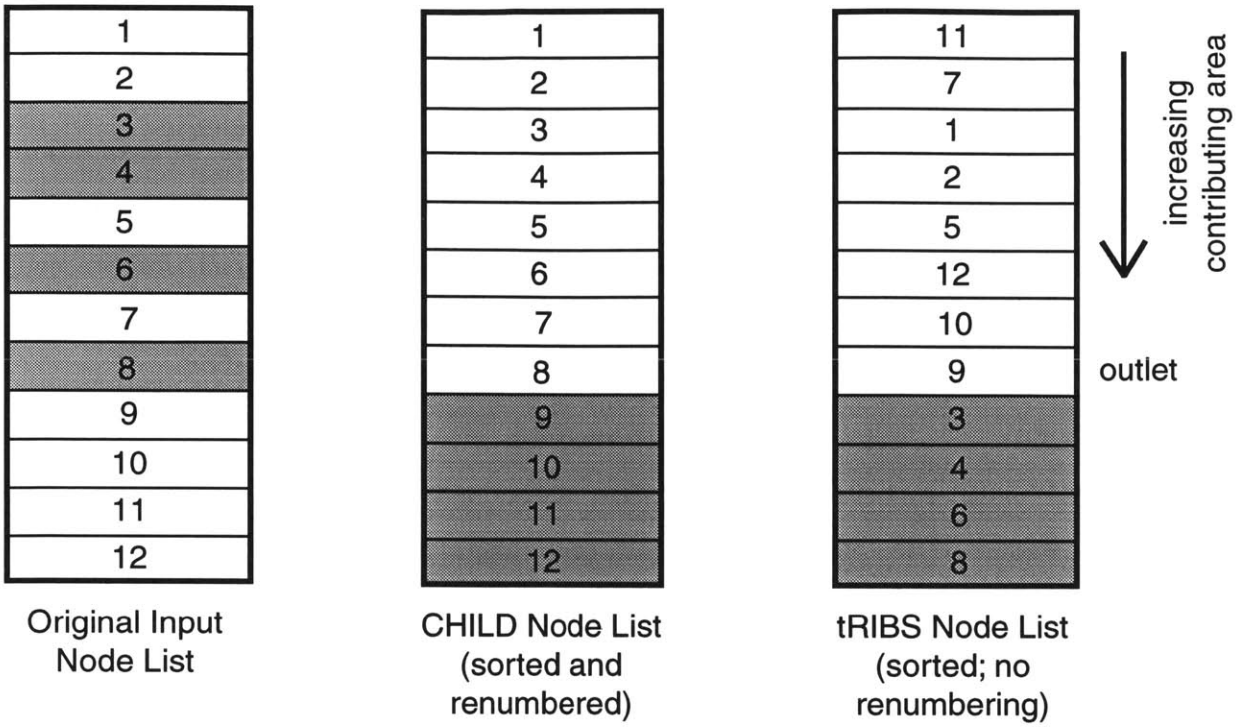


Figure 3-1: An example of the differences between a CHILD and tRIBS node list. Boundary nodes are shaded in gray.

ous rainfall record, limiting the required data storage space in tRIBS. The complete code for tRainfall is included in Appendix B as tRainfall.cpp and tRainfall.h.

For efficiency, the original CHILD model renumbers the node list as nodes are added/removed from the system. To allow for a system with time varying inputs a node must keep the same ID number for the entire model run. This does not mean that the node list in the system is kept static. Nodes are ordered based upon their contributing area, with upstream nodes near the top of the list and the outlet toward the end of the list. Boundary nodes are also included at the very end of this list. The changes made for tRIBS stop the renumbering process in the model, and an example showing the difference between the two node lists can be seen in Figure 3-1.

The system of runoff production was completely revamped in tRIBS. The tStreamNet object given in the CHILD model was defined for use in a sediment transport model. In this object, items that are not found in RIBS, like layering and meandering channel networks, caused significant problems. A new object, tFlowNet was created using some of the basic functions of tStreamNet including a pit filling algorithm and

a procedure for sorting nodes by network position in upstream to downstream order [74]. In this scheme, the headwaters of a basin are put at the top of the node list and the outlet is shuffled to the end using a cascade algorithm [9]. New functions were added to tFlowNet to incorporate the surface flow routing methods of RIBS as described in Chapter 2. These include functions defining the travel time for each node and methods for determining flow at the outlet after completion of each time step in the simulation. The computer code for the new object, tFlowNet, is presented in Appendix B as tFlowNet.cpp and tFlowNet.h.

A new saturated groundwater system was added to CHILD to replace the one present within RIBS. Following the work of Jackson [40], the lateral subsurface flows were defined using a multiple direction (2–D) flow model. With modifications to allow for the use of voronoi cells in the flow model, one can define the volume of inflow/outflow from each node, i, as

$$V_i = \left[T \sum_{j=1}^{n} S_{ij} \lambda_{ij} \right] dt \tag{3.1}$$

where n is the number of nodes directly adjacent to i, S_{ij} is the slope of the groundwater surface from i to j, dt is the model time step and λ_{ij} is the width of the shared voronoi edge. The transmissivity, T, is calculated using the assumption that hydraulic conductivity in the unsaturated zone decreases exponentially with depth as presented in Chapter 2. Using this assumption, transmissivity is defined as

$$T = \frac{K_{sat}}{f} e^{-fNwt}. \tag{3.2}$$

Inflow/outflow volume for each voronoi cell is then computed and the change in water table height is given by

$$Nwt_i = Nwt_i + \frac{V_i}{\Lambda_i} \tag{3.3}$$

where Λ_i is the voronoi area of node i. This flow model has been tested for various time steps (30 seconds to 1 hour) and various voronoi cell sizes ($400m^2$ to

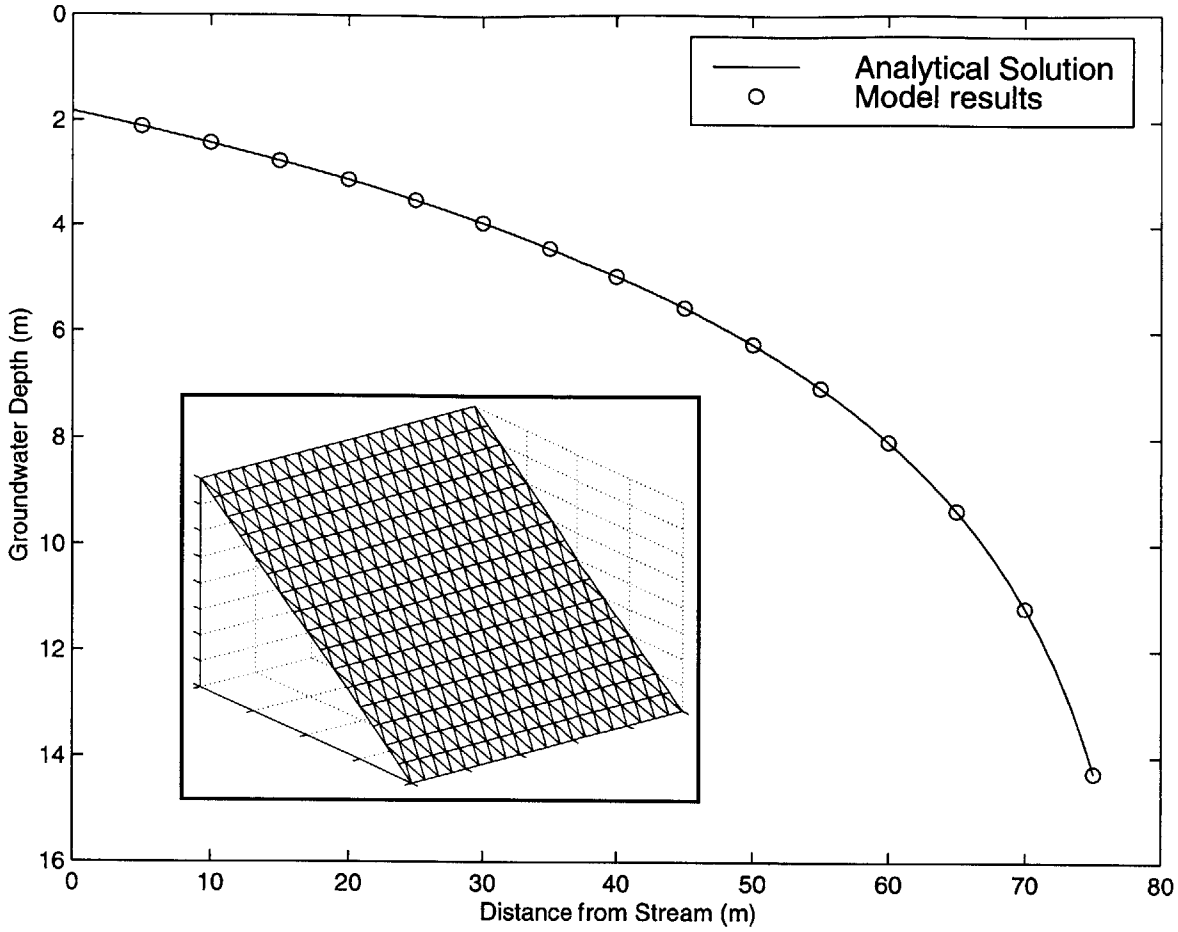


Figure 3-2: Numerical solution to the 2–D groundwater flow equation compared with a 1–D analytical solution for a simple hillslope(inset).

$10000m^2$) with no stability problems. Figure 3-2 shows a comparison of an analytical steady state solution for a 1–D hillslope under constant rainfall and the steady state numerical solution of the model.

Other changes to the CHILD model were needed to provide a framework for RIBS. The program that creates the watershed structure, tMesh.cpp, was modified to allow for a new node type. Only outlet, boundary, and interior nodes were allowed in CHILD originally. For tRIBS, a new node type was added to identify stream nodes in the watershed. Changes to tNode, tList, and tNodeList were made to support this change. Many areas of the CHILD model code outside the objects mentioned earlier were also dependent upon objects not associated with the basic framework of CHILD. These dependencies were removed to create the tRIBS model.

3.2 Changes to the RIBS model

Many changes were needed to convert RIBS from an event based to a continuous flood/forecasting model. The work to complete this task was done by Valeri Ivanov as part of a coordinated effort to create tRIBS. The changes described in the paragraphs to follow are his, and are described here to provide a complete picture of the tRIBS model. The concepts presented here will be more fully described in his own work which will be published within the next year. [39]

Significant changes have been made to some of the basic assumptions in the original RIBS model. Specifically, the infiltration process in tRIBS is now defined by kinematic infiltration and capillary suction. A modified Green-Ampt model developed for non–uniform soils has been incorporated into tRIBS and capillary suction is calculated using a 1–D Darcy formulation of unsaturated flow developed by Neuman [52]. Modifications to this formulation were made to account for the exponential decrease in conductivity assumed in the RIBS model.

The calculation of this new infitration rate begins with the Green-Ampt formulation given as

$$i = -K_{sat}\frac{(h_f - h_o - N_f)}{N_f} \tag{3.4}$$

where i is the infiltration rate, h_o equals the depth of ponding and N_f equals the wetting front depth. The expression for water pressure head, h_f, has been developed by Ivanov [39], and is given by

$$h_f(\theta_a, \theta_b, z) = -\frac{1 - e^{fz}}{fz} \cdot \frac{\Psi_b \left[S_{eb}^{\frac{1+3\lambda}{\lambda}} - S_{ea}^{\frac{1+3\lambda}{\lambda}} \right]}{\lambda \left(3 + \frac{1}{\lambda}\right)} \tag{3.5}$$

where λ is the pore–size distribution index, θ_a and θ_b are arbitrary soil moisture values, Ψ_b equals air entry bubbling pressure, and S_{ei} equals

$$S_{ei} = \left(\frac{\theta_i - \theta_r}{\theta_s - \theta_r} \right) \tag{3.6}$$

Following the work of Smith [69], this new capillary suction is used only when the soil moisture level in a pixel is below a set threshold. Above this threshold, the RIBS kinematic model described in Chapter 2 is followed. This new formulation does add one new parameter to tRIBS as an air entry bubbling pressure must be given for this new infiltration process.

One other parameter has been added to tRIBS. In moving to a continuously simulated model, one must explicitly define the beginning and end of a storm event. This is essential in tRIBS as multiple wetting and top fronts are not allowed. The parameter, IntStormMax, is used for this purpose as it defines a maximum dry period after which a new storm event is assumed. If a dry period is less than IntStormMax, the wetting and top fronts from the previous rainfall event are continued. If a dry period is greater than IntStormMax, a new storm is assumed. In that case, any existing wetting and top fronts are redistributed throughout the unsaturated zone and new fronts are created.

Interstorm conditions must be modeled in the tRIBS model and six new pixel states have been added to accomplish this task. The first new state, WTDrop, calculates the results of a required water table drop from the surface. In this situation, an initially saturated pixel has a negative moisture flux, requiring a lower water table.

Another new state, interstorm conditions, occurs when the amount of moisture remaining in the unsaturated zone is less than the residual moisture profile required for a particular water table depth. To correct for this situation, a lower water table is computed to equal the available moisture in the unsaturated zone. The wetting and top fronts are moved to the water table in this state unless IntStormMax is reached. In that case, the wetting and top fronts are returned to their initial value of zero.

Storm to Interstorm transition takes place in many pixels immediately following a rainfall event. When the recharge is less than zero, but moisture content is still greater than the residual level prescribed by the water table depth, the pixel is transitioning from a storm to interstorm condition. The wetting and top front continue to move downwards in this case, but the moisture content in the pixel decreases, creating smaller, less saturated edges along these fronts. This state can also occur when

recharge is less than infiltration in a surface saturated pixel.

During significant rainfall events, the wetting front may reach the water table. When this occurs, the original RIBS model would redistribute the moisture raising the water table and a new wetting front would begin at the surface. This can lead to a rapidly varying wetting front profile with numerous discontinuities during a model simulation. Also, this new wetting front will descend faster than the initial front due to increased soil moisture and decreased water table depth. To avoid this erratic behavior, the new model does not create an additional wetting front after the first front reaches the water table. All additional water entering the pixel is used to raise the water table. Figure 3-3 compares the evolution of the wetting front when it reaches the water table in the new formulation to the erratic evolution of the old formulation. In the model this is called the "storm evolution" state.

Two other pixel states, WTExactInitial and WTStaysatSurface are also included for completeness in tRIBS. WTExactInitial only occurs when the moisture in the unsaturated zone is equal to the initialized state of the pixel. This state requires no computation as all the variables are left exactly the same. In WTStaysatSurface, all excess lateral flows and any additional recharge are accounted for as runoff to the outlet. The application of each of these six pixel states can be seen in the Cbsim object presented as cbsim.cpp and cbsim.h in Appendix B.

While not directly part of the tRIBS model, a new initialization scheme for the ground water level was formulated. Following a method developed by Sivapalan [66], one assumes that a reasonable approximation of the initial water table can be obtained by stating that the recession discharge prior to a storm, Q_o, results from a quasi steady rate of recharge to the water table. Using the definition of a pixel for RIBS given in Chapter 2, an equation for initial water table depth is calculated as

$$Nwt = \frac{-1}{f} ln \left[\frac{afQ_o}{AK_{sat}a_r W S_i} \right] \tag{3.7}$$

where a represents the total area drainage through a pixel per contour length, A equals the watershed area, W is the pixel width, and S_i equals the surface slope. The

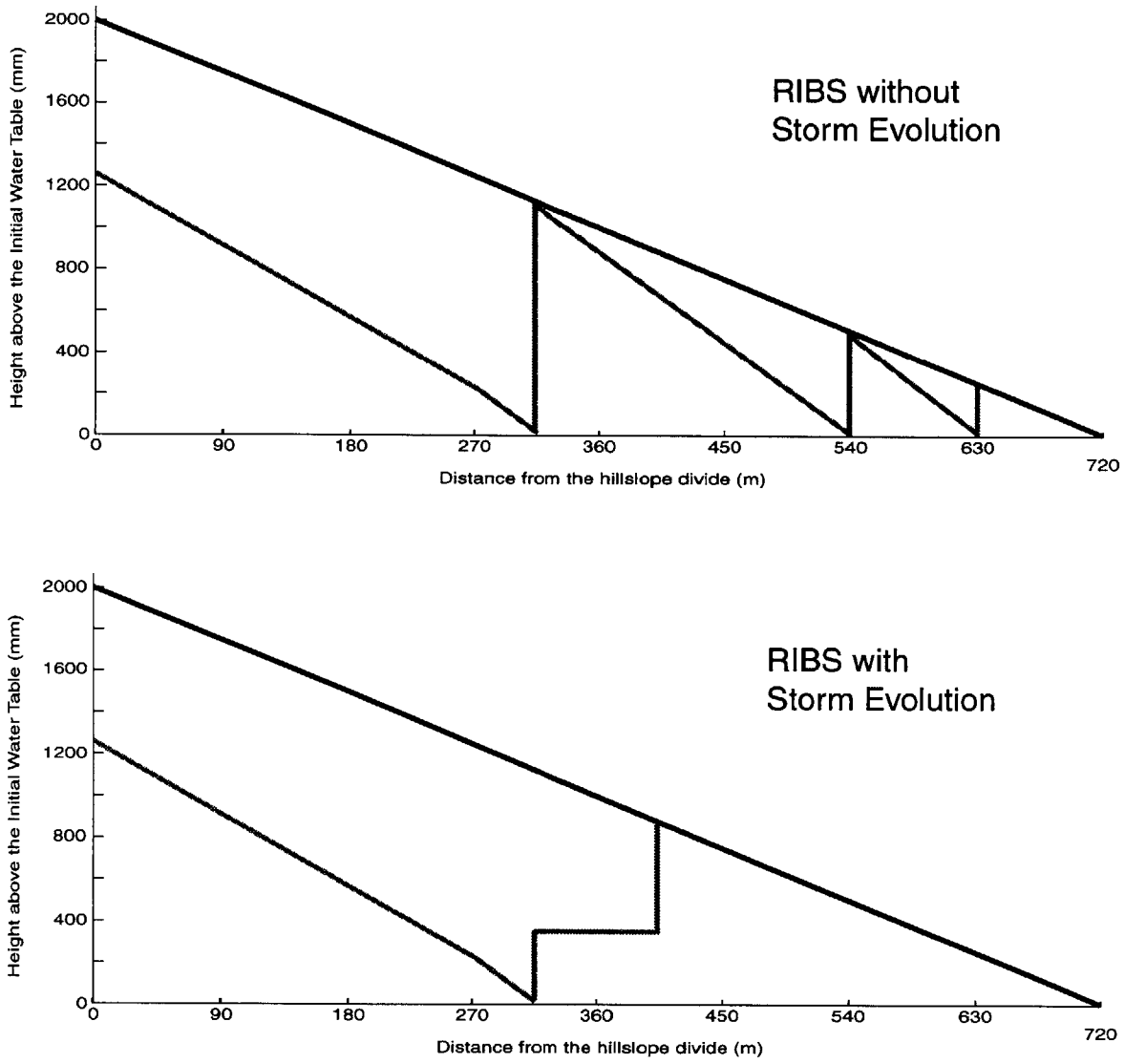


Figure 3-3: An example of the behavior of RIBS with and without the new storm evolution pixel state. The surface is given by the solid black line, and the wetting front with a solid gray line.

use of this approach without modification can result in a wildly varying water table depth due to rapid changes in slope and abrupt changes in upstream contributing area in neighboring pixels. These rapid changes are uncommon in nature, and a smoothing process is used to correct the water table depths.

Chapter 4

Collecting and Manipulating Distributed Data

Topographic and rainfall data were collected and manipulated using a GIS (Geographic Information System) to create data inputs for the model. Other data sets were also used to support the model including soils, streamflow and groundwater well data. These collected and manipulated data sets have been published on a public web site located at:

http://platte.mit.edu/watdata/-watdata.html

With this web site, these data sets are available to the entire hydrologic community.

4.1 Topographic Data

The USGS Digital Elevation Model (DEM) data sets were used for this project. As part of the National Mapping Program, 7.5 minute, 15 minute, 2 arc-second, and 1 degree DEMs have been produced and are available through the USGS for the entire United States. DEMs at the 7.5 minute scale will be used here as they are largest scale available. Also, these DEMs correspond directly to USGS 1:24,000 quadsheets. This makes finding the proper DEM easy, as one can first locate the target watershed using paper copies of quadsheets found quite readily in most university libraries. Two types of 7.5 minute DEMs exist, one with a horizontal resolution of 10m and the other with

a resolution of 30m. The 7.5 minute DEM map series with a horizontal resolution of 30m will be used here for two reasons. First, over 90 percent of the US is covered by the 30m series. Second, this data is available at no cost on the USGS FTP server while the higher resolution 10m data is not.

The array created in a 7.5 minute DEM will not be a standard rectangular array. Due to the curvature of the earth, each column does does not have to have the same number of rows and the left and right sides of the quadsheet will not be parallel. (Figure 4-1) This curvature of the earth leads to the use of a geographic projection to create a flat DEM. The USGS uses a Universal Transverse Mercator (UTM) map projection to accomplish this. The UTM projection coordinate system is further defined with a datum which will be used to provide the zero point for the coordinate system. In USGS DEMs, either the North American Datum (NAD) 27 or NAD 83 is used along with an elevation datum. Elevations are given relative to the National Geodetic Vertical Datum of 1929 (NGVD 29) and can be given in feet or meters. In most cases, quadrangles with steep topography (those defined by contour intervals of less than 10ft) are given as elevations in feet. Areas of lesser topographic slope are given in elevations of meters [70]. This is an important point of note as adjacent DEMs may use different units for measuring elevation.

It is important to know the expected accuracy of this data product. DEM data for 7.5 minute units have been produced using four different methods and have three classification levels. Much more detail on these methods and classification levels can be found in the specification guides produced by the USGS [70]. All of the DEM data in this project uses a method which interpolates digital line graph contour data. All DEM data sets used here have been classified as Level 2 data products, which means that the elevation set provided has been processed and smoothed to remove systematic errors in the dataset. Under Level 2 specifications, a maximum root mean square error (RMSE) equal to one half of the contour interval is all that is allowed when defining the surface elevation, and this is checked by looking at the differences between the "true" (based upon benchmarks) and interpolated values of 28 points (20 interior and 8 edge points) in each DEM. A typical RMSE seen in this project

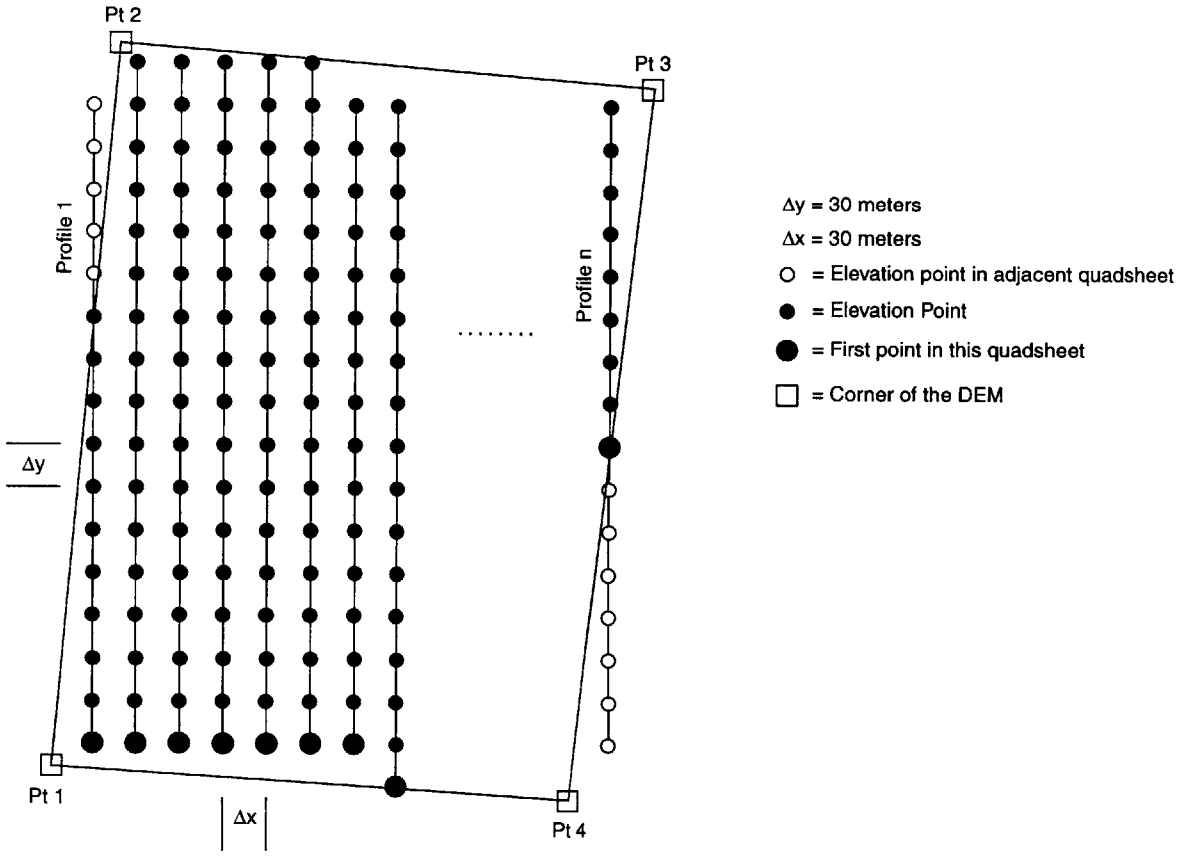


Figure 4-1: A schematic of a quadsheet represented as a DEM

was approximately 4m which is well below the required tolerance.

4.1.1 Converting DEM Data

USGS DEMs can be freely downloaded from their web site at:

http://edc.usgs.gov/doc/edchome/ndcdb/ndcdb.html

The list of DEMs can be searched by quadsheet name, state, or even by using an interactive graphical user interface available at the web site. Since a quadsheet normally represents 160 square kilometers, this process will be repeated numerous times as only very small watersheds will fit on one quadsheet.

One caveat in using the free USGS DEMs is that they are only available in SDTS (Spatial Data Transfer Standard) format. Ratified by the National Institute of Standards in 1992 as FIPS 173, SDTS has become the mandatory format for all federal spatial data developed after 1994 [38]. Unfortunately, much of the GIS and database community has ignored this standard for too long. Early in this project, an Apple Macintosh program developed by Sol Katz at the Bureau of Land Management was used to convert SDTS DEM data into ASCII grid data. This data could then be moved into ARC/INFO using their ASCII to grid conversion routines. Recent SDTS conversion has been completed with simpler routines which have just recently been developed. ESRI has implemented SDTS conversion routines in the latest version of ARC/INFO and a group of researchers at Utah State University [29] have created an ArcView routine to automatically import SDTS DEM data. Figure 4-2 shows a 7.5 minute, 30 meter DEM converted into ARC/INFO grid format.

With a full set of converted DEMs, one must make sure each adjacent DEM actually matches the rest of the set. Projections (watersheds often cross UTM zones), datums, and vertical units of measure should be modified as needed to present a consistent set of DEMs. Work then must be completed to assure that edges of adjacent DEMs match properly. In theory, all DEMs should fit together neatly like a jigsaw puzzle. This often doesn't happen, most likely due to errors in initial data processing by the USGS [29]. A gap of one to two pixels is very common, and must be filled in before watershed delineation can occur (See Figure 4-3). An ARC/INFO script

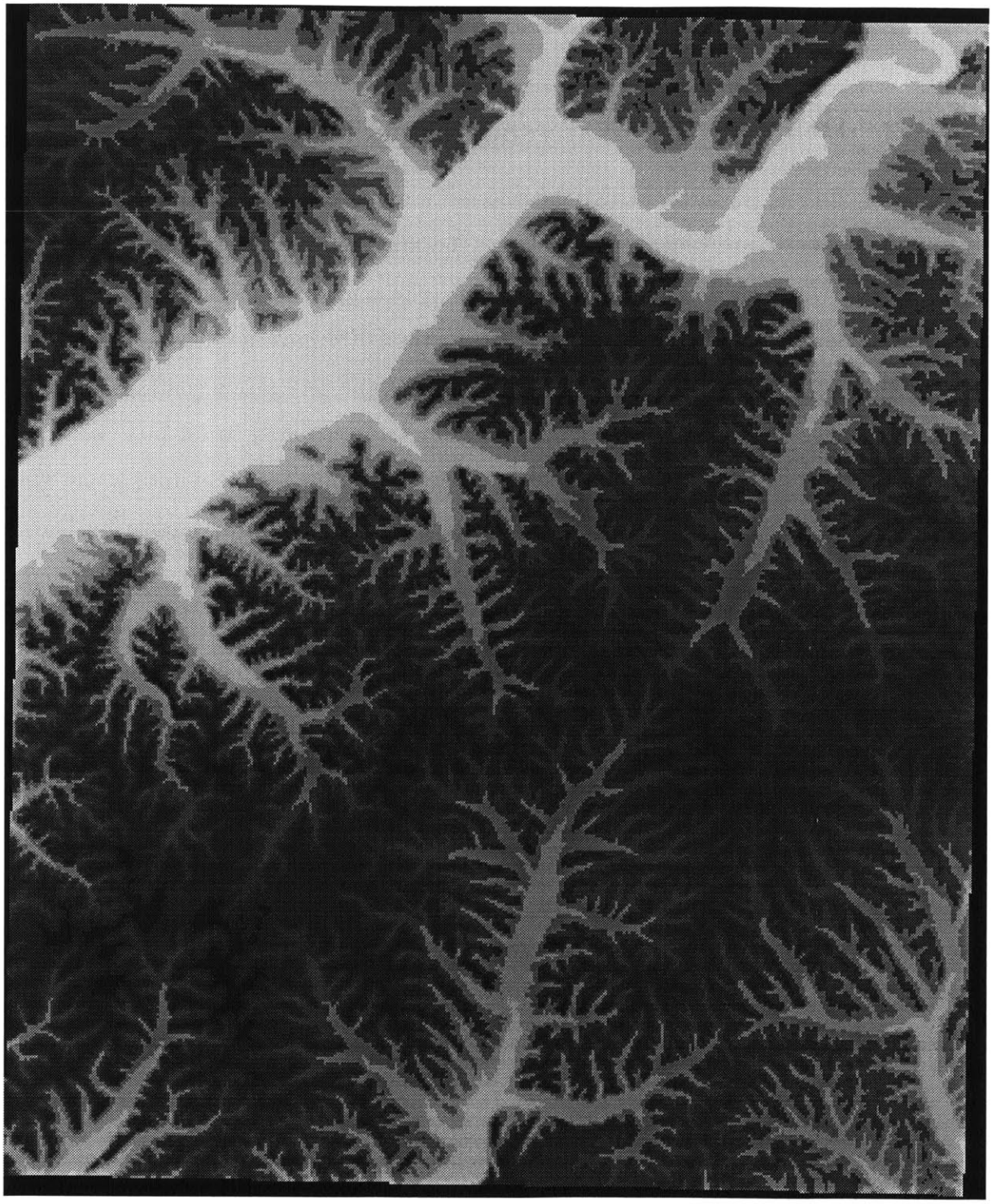


Figure 4-2: A 7.5 minute, 30m DEM for Chewey, Oklahoma

written in AML (Arc Macro Language) was created to fix these single cell gaps by averaging across the gap. The AML code used to complete this task can be found in Appendix B.

4.1.2 Delineating a Watershed using Topographic Data

With a complete set of DEMs for the selected watershed (Figure 4-4), one can begin the process of delineating the watershed. The first step involves filling the 'pits' in the DEM. One of the basic assumptions in hydrologic modeling is that all pixels drain to the pixel downhill from that pixel. Pits in the DEM have no downhill neighbor, and trap water in the system. As the natural landscape very rarely contains pits, one can safely say that these pits are errors in the data set [53]. To fill pits, we use a fairly simple algorithm which increases the elevation of any pit pixel until it can flow downhill. This is completed using a fairly standard algorithm developed by David Tarboton [73] and implemented within ARC/INFO. This pit filling routine does have some problems in very flat terrain where the change in elevation between pixels is close to the RMSE [45], but the basins chosen for this project have enough vertical relief to avoid this problem.

While filling the pixels, flow directions for each of the pixels is computed as part of the process. In this model, we use a unidirectional model of surface flow. That is, flow only goes in the direction of steepest topographic slope. Others have developed multi-directional flow models (e.g.,[15];[72]), but in watersheds with tens of thousands of pixels, missing one or two pixels when delineating the watershed is not significant enough to warrant the use of a more complex algorithm. With the flow directions computed, the amount of flow through each individual pixel can be found. This is known as the contributing area of a pixel and can be found by adding up the flow that goes through each and every pixel. A recursive algorithm for this was developed by David Mark [46] and it has been used through its implementation in ARC/INFO.

At this point the watershed can be created. With a known gage location for the outlet of a particular basin, one can use the flow directions and contributing area to find the watershed boundary. The flow directions are used to find the boundary of

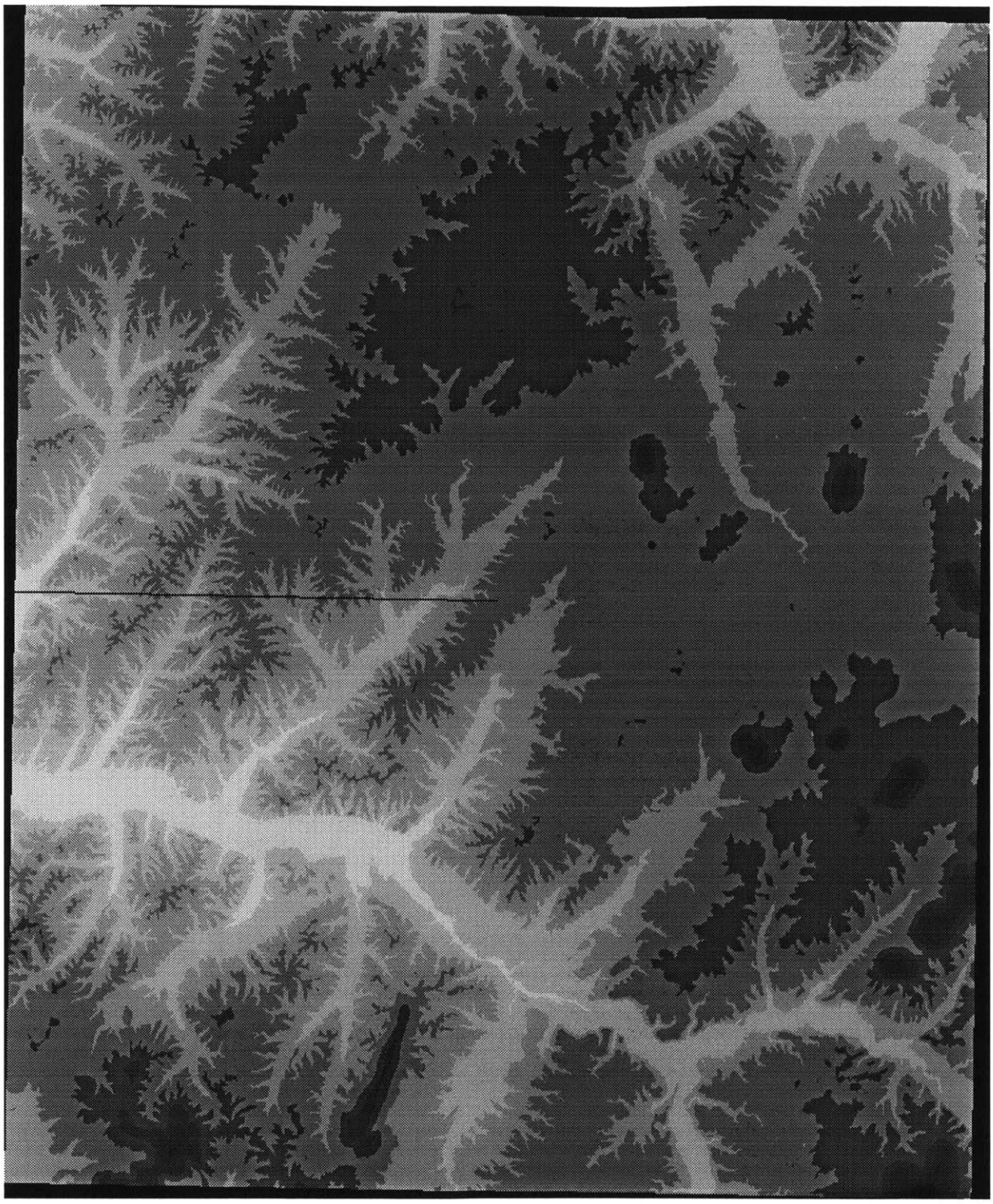


Figure 4-3: Four DEMs in Eastern Oklahoma. The black line running from left to right in the center of the page is a gap between two DEMs.

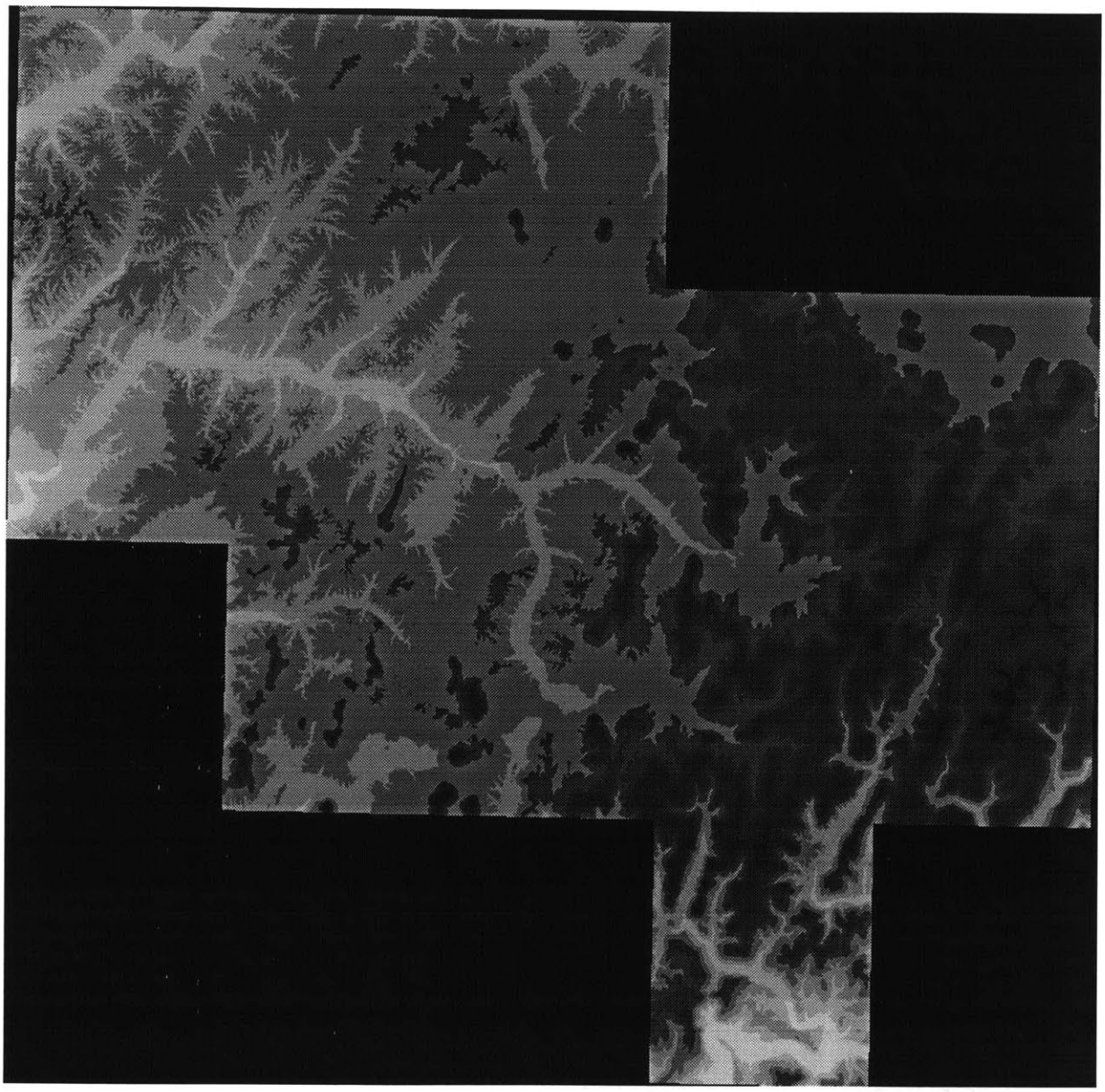


Figure 4-4: A complete set of merged DEMs. These will be used to find the Baron Fork Watershed

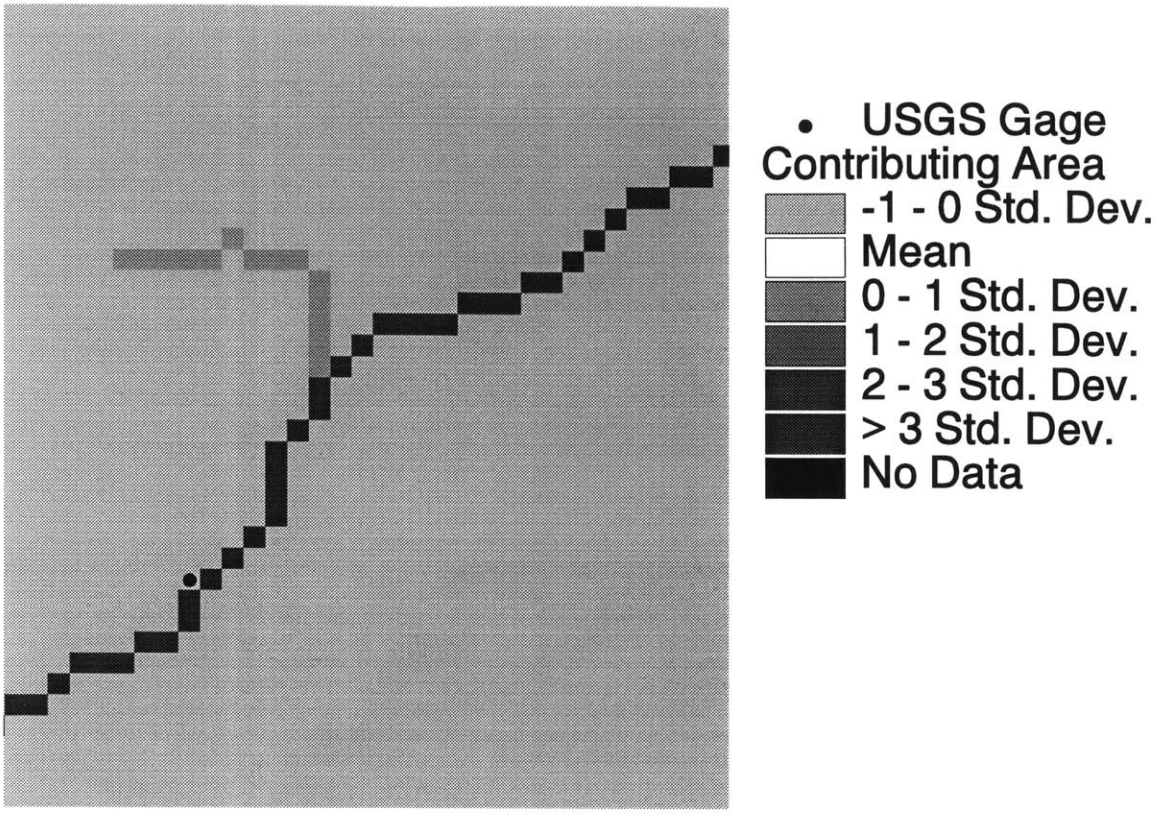


Figure 4-5: Contributing Area and USGS Gage Location for the Baron Fork Watershed, OK

the watershed and the contributing area helps one find the outlet of the basin you are using. Often times, the location of the given gage will be off of the location of the true outlet of the basin as shown by the contributing area. As shown in Figure 4-5, the gage and contributing area near the outlet of the Baron Fork watershed do not match. If one blindly used the gage location as the outlet in the case of shown, a one pixel watershed would be the result. In using the contributing area to find the basin outlet, one gets an accurate representation of the watershed. The final watershed map of surface elevation is now complete, and the final watershed can be seen in Figure 4-6.

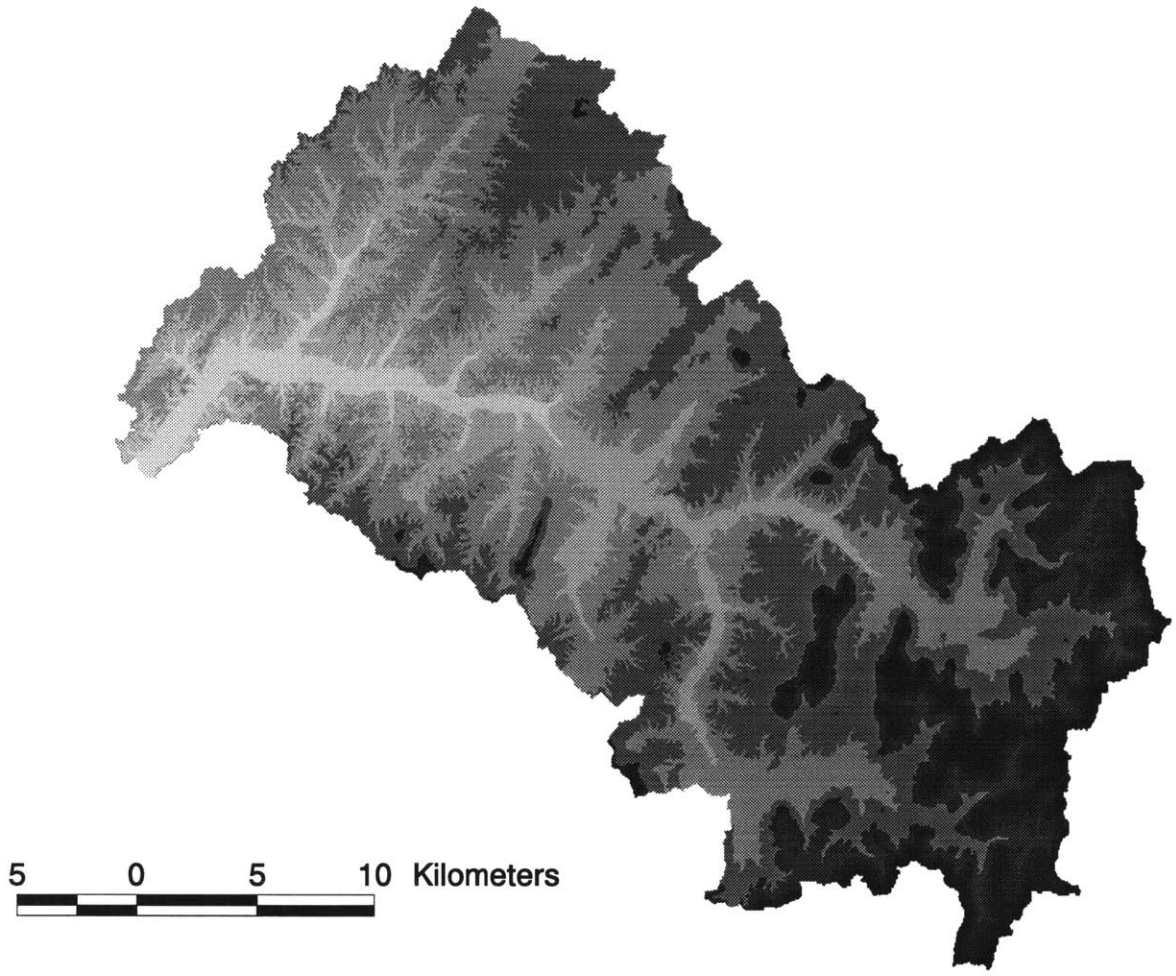


Figure 4-6: The Baron Fork Watershed, OK

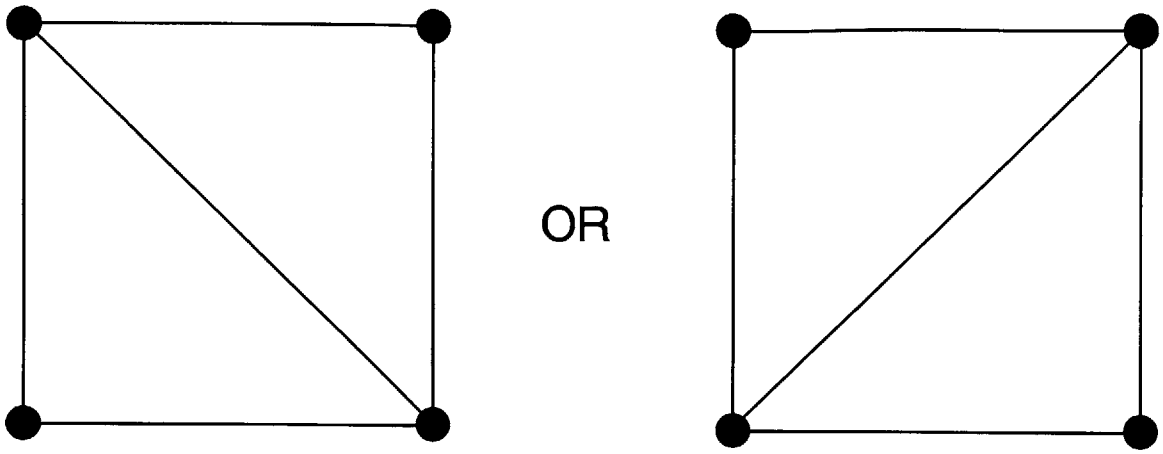


Figure 4-7: Non-unique diagonal found with four equally spaced points

4.1.3 Creating the TIN

At this point, the watershed is still represented in a regular grid structure. The tRIBS model uses TINs, so a transformation must occur. One major problem is that a regular grid creates poorly defined triangles. In a regular grid with equally spaced points, the diagonal created with any four points is not unique. The diagonal can be at two places (see Figure 4-7), both of which completely fit the definition of a TIN presented in Chapter 3. There is not a unique TIN to fit this set of four points. Also, if every point is the grid was used, the TIN would be too complex computationally. Therefore, a thinning process is first performed on the grid.

This process of thinning the grid can be performed in numerous ways. The VIP (Very Important Point) process implemented in ARC/INFO was used here. The VIP process selects points based upon their significance in describing the surface topography of the watershed. This process works on a local scale, selecting points based upon the magnitude of the second derivative (curvature) at each point in the grid. The process algorithm works by checking the four transects of each point created by its eight neighboring grid points. The difference in elevation between each transect and the center point measures the significance of that center point (see Figure 4-8). This only provides the relative significance of each point. To then thin the grid, one chooses a percentage of points to be saved in each the grid. This is a very subjective value, and is chosen by the user to fit their own personal needs. For this project,

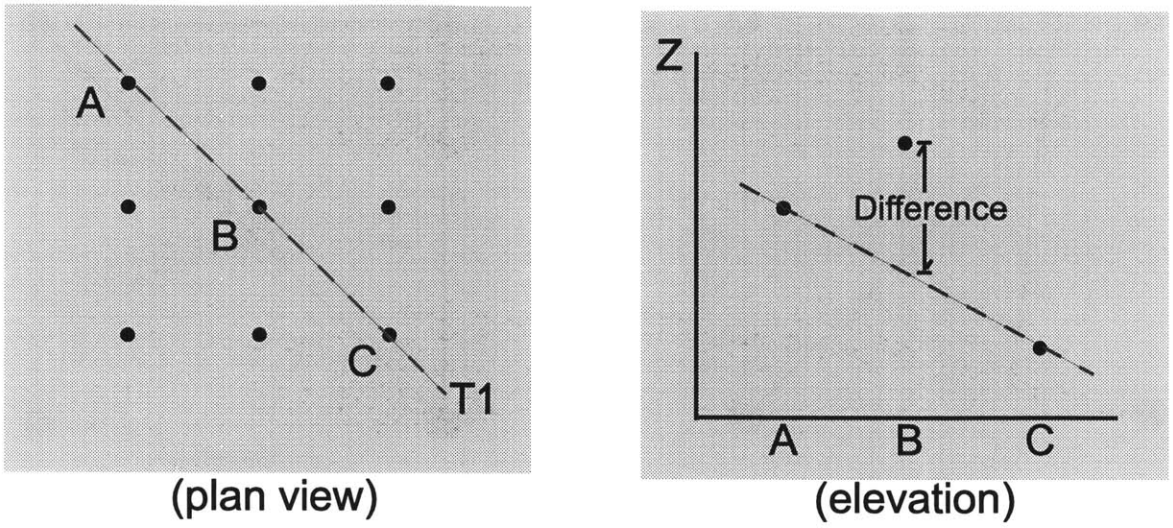


Figure 4-8: The VIP Process

it was found that keeping as little as 2 percent of the original grid still provides accurate representations of the watershed. An example of a watershed created under this process is shown as Figure 4-9. The TIN can be further modified by adding the streams to the TIN. Using a set of stream networks created by the US EPA and USGS (see section 4.3), one can add new points the to the TIN. A TIN created with an imbedded stream network is given as Figure 4-10. In Figure 4-10, one can see an increased number of voronoi cells along the stream channels in comparison to the TIN created without the stream network (Figure 4-9).

To prepare the TIN for use in the tRIBS model, special care must be taken in dealing with the boundary of the TIN. Since the accumulated area of the voronoi cells, not the total area of the TIN, defines the watershed area, a direct use of the TIN developed earlier will not give a correct representation of the area of a watershed. Figure 4-11 shows this well with the area in red showing the area that would be neglected due to this fact. Also, voronoi areas at the boundary have a tendency to become long and narrow spikes due to the way they are constructed. The red area in Figure 4-12 is a common representation of what happens in this situation. To avoid these two problems, a double ring of points is added which mimics the boundary, one ring added inside the original boundary, and one outside the original boundary. Both rings are offset an equal distance from the original boundary, in this case, 90

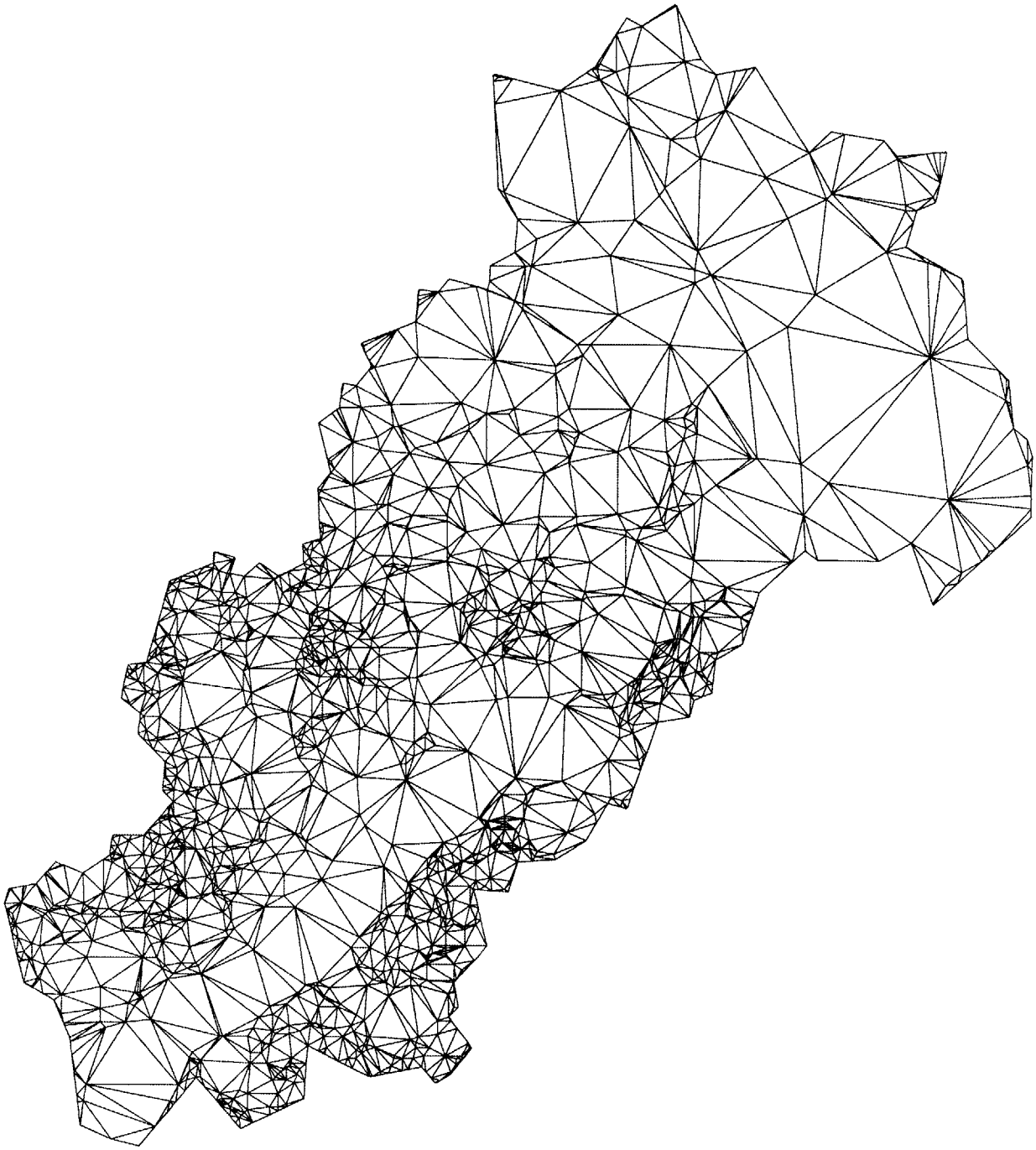


Figure 4-9: TIN of Peacheater Creek, OK

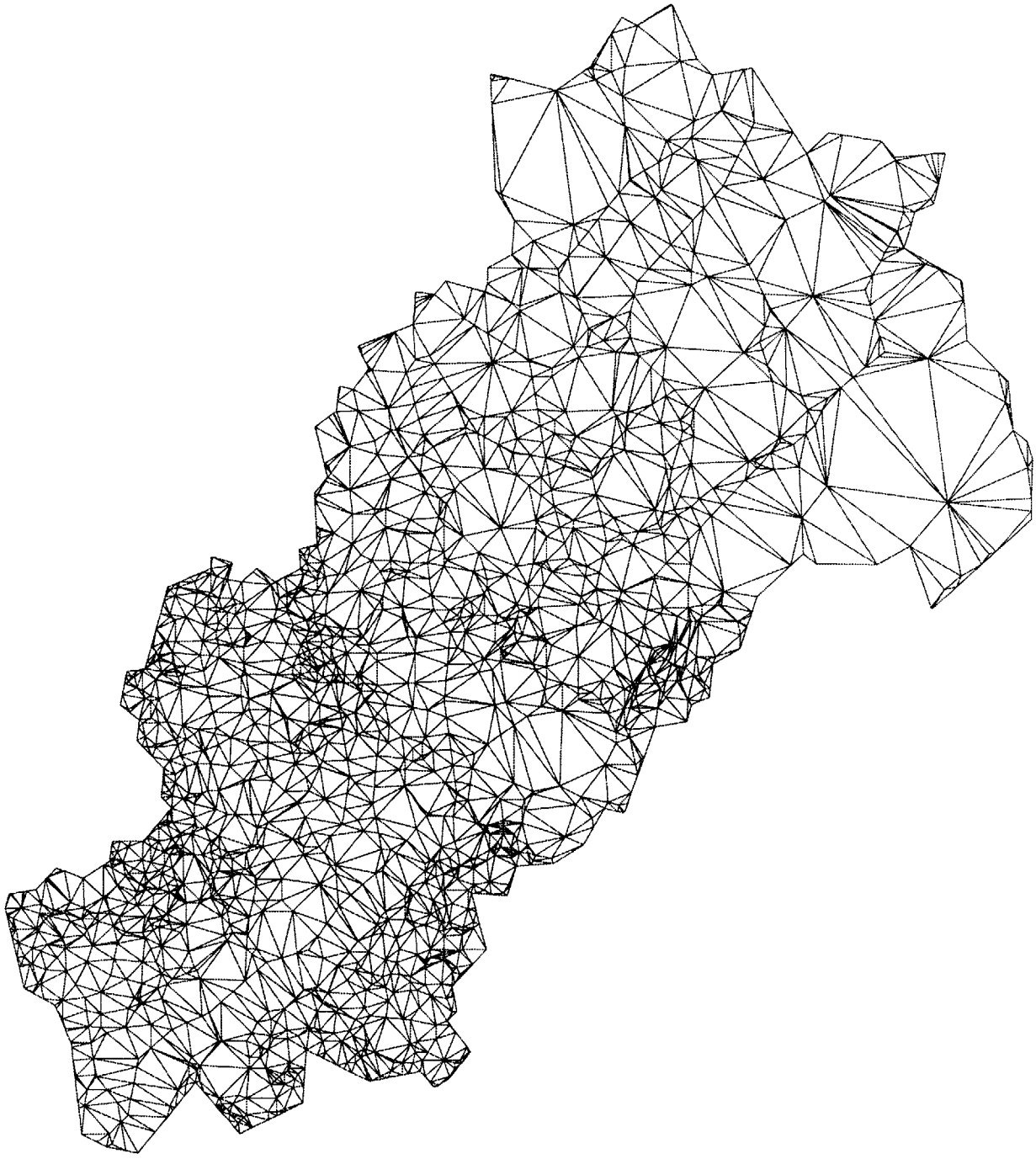


Figure 4-10: TIN of Peacheater Creek, OK with the addition of an imbedded stream network

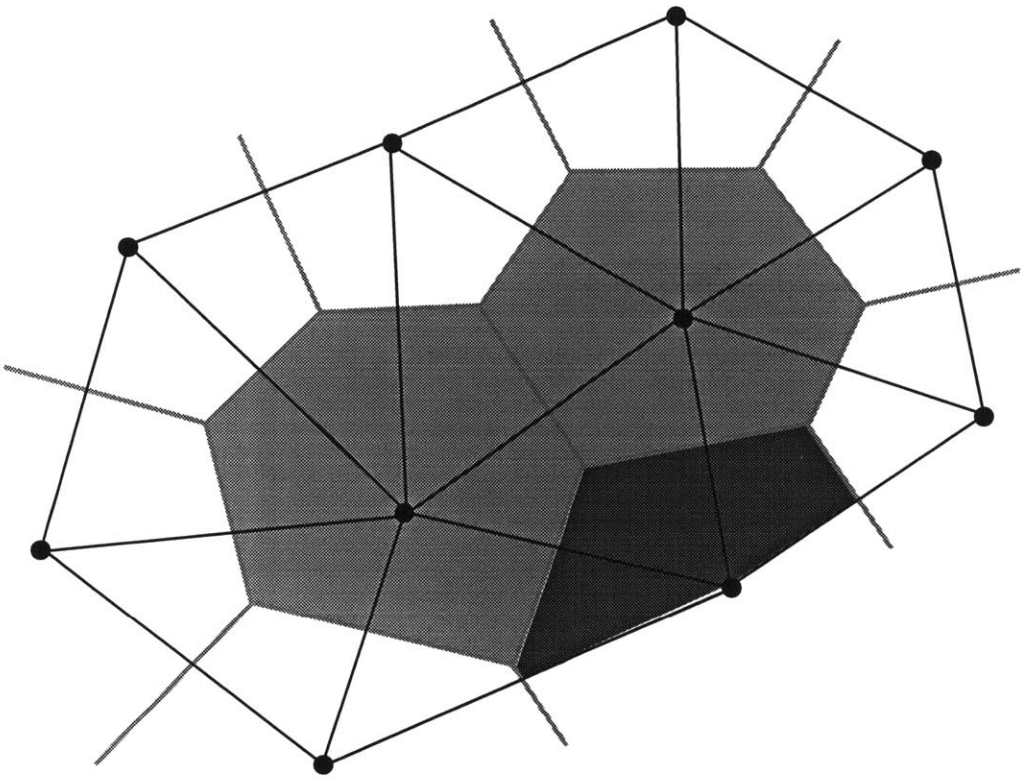


Figure 4-11: Missing Voronoi Areas along the edge of the TIN Model

meters. The original boundary is then eliminated, and a new TIN is created using these points along with the original interior points used in Figure 4-10. This new TIN avoids the problem of fitting voronoi area as the new voronoi area will closely match the original boundary of the watershed. This can be seen in Figure 4-13. With two rings of data points along the boundary (lines A and B) replacing the original boundary, the problem of incorrectly determined voronoi areas is reduced drastically as the shaded area in Figure 4-13 is now part of the watershed. The final TIN used in the model is now complete and the finished product can be seen in Figure 4-14.

4.2 Rainfall Data

In comparing two different models, one wants to make the test as fair as possible. For this reason, the National Weather Service (NWS) Next Generation Weather Radar (NEXRAD) was used as the rainfall data set. This data is already used in the Sacramento model and it make sense to use this same data in the tRIBS model. The

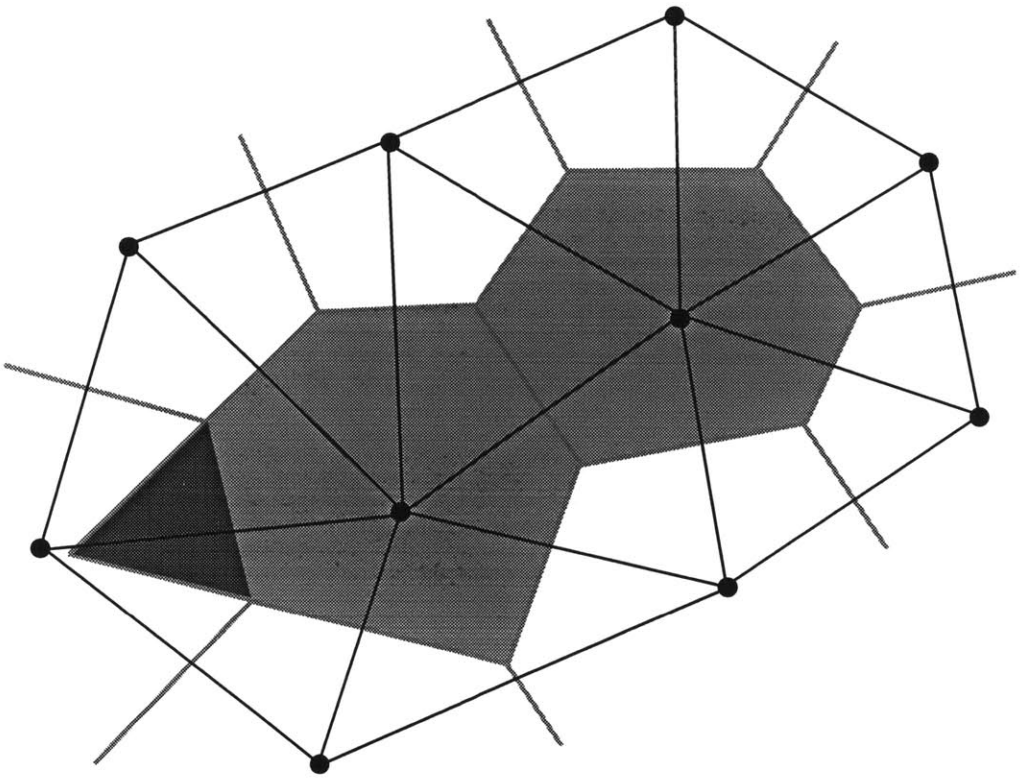


Figure 4-12: Spikes formed along the TIN boundary

NEXRAD data is produced hourly and is given on a grid which is approximately 4 by 4 km in size.

Three stages of processing exist for the NEXRAD data product. The Stage I product presents an estimated hourly rainfall amount for a single radar site using only the raw radar reflectivity. The reflectivity is transformed using a reflectivity to rainfall relationship, otherwise known as a 'Z-R' relationship. This Stage I product is rarely used as known errors exist in the rainfall estimates [67]. Using gage measurements as a "true" data value, the Stage I product is modified in an attempt to correct for these errors. This product is NEXRAD Stage II. The final data set combines Stage II data from many overlapping radar stations into one final product [68]. This combined data set is NEXRAD Stage III, and it is the data set used in all future modeling efforts shown later.

Two types of errors can effect the NEXRAD Stage III Data. The first type of error occurs when the estimate of rainfall magnitude is incorrectly measured. The

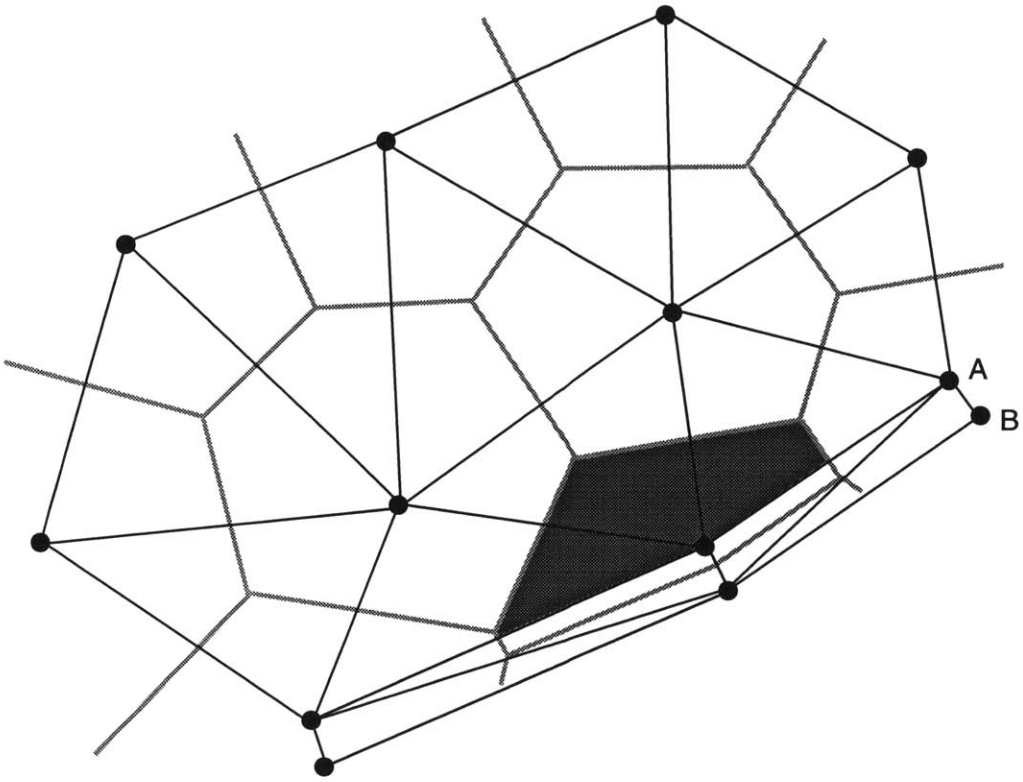


Figure 4-13: Corrected voronoi areas along the watershed boundary using the double ring method

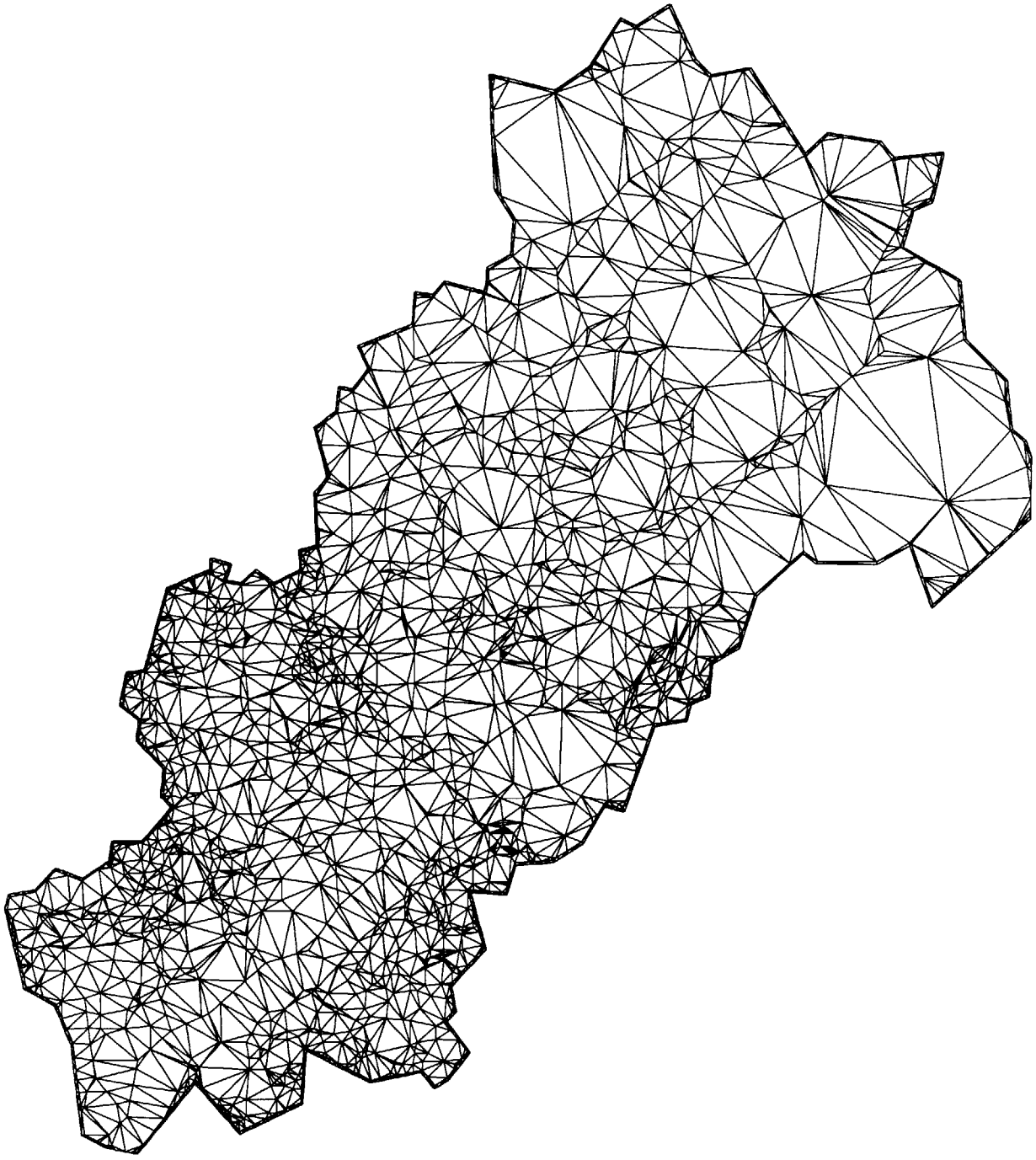


Figure 4-14: Final Complete TIN for Peacheater, OK

Z-R relationship is not exact, so errors in transforming reflectivity to rainfall do occur with some kind of systematic bias [67]. Also, radar is notoriously bad in mountainous areas under cold weather conditions where precipitating clouds often form below the radar beam avoiding detection [80]. These last two conditions are significant, but can be controlled with careful selection of the study basins used in the modeling exercise.

Another error which needs to be mentioned is a locational error in the NEXRAD Data. NEXRAD data is collected under the HRAP coordinate system originally developed by the Navy. This is a spherical representation of the earth which works well for the GCM simulations it was developed for. Unfortunately, the earth is not really a sphere, but more of an ellipsoid. At the regional scale that dominates hydrologic flood-forecasting, one cannot ignore this elliptical shape. In fact, all of the other data in the project are already mapped to an ellipsoid. Therefore, one needs to transform data from the sphere to an ellipsoid which will cause a locational error in the data. In the latitude ranges of the continental United States, this error will vary from approximately 0.3 % at 48°N to 0.6% at 25°N latitude. [59] This transformation will be discussed in more detail below.

4.2.1 Converting NEXRAD Data

Through a collabrative agreement with the NWS Hydrologic Research Lab (NWS-HRL), the NEXRAD Stage III data was obtained. The data is formatted in xmrg binary format which has undergone constant redefinition during the years studied here. Different headers exist based upon the time that the particular radar set was created. Converting these various xmrg files to useful ASCII data is completed using an algorithm developed by Seann Reed at the NWS-HRL.

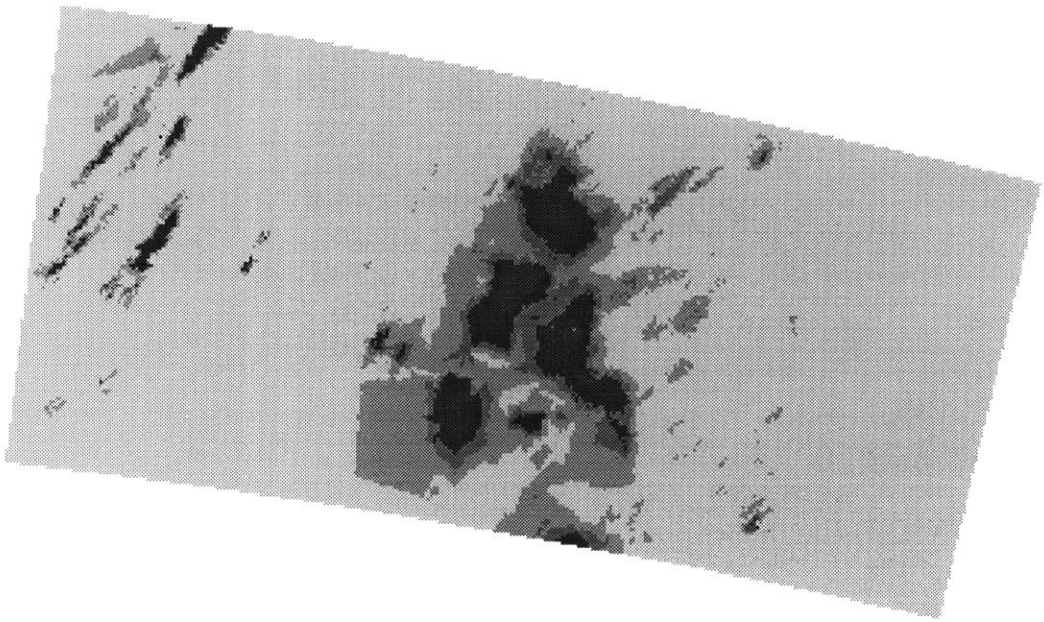
The data must now be projected to match the geographic coordinates of the topographic data. The first step involves defining the initial projection used in the HRAP coordinate system. The standard spheroid used by ARC/INFO does not match the spheroid used in the HRAP coordinate system. Due to this fact, a mathematical trick is used to get the correct geographic location. As proposed by Tom Evans at the U.S. Army Corps of Engineers, the latitude datum of the HRAP coordinate system is

slightly shifted to correct for the spheroid differences. The HRAP system is technically defined with a starting point of 105°E and 60°N. By moving the latitude slightly to 60°, 0 minutes, and 24.5304792 seconds N, one can then use the standard polar coordinate projection given in ARC/INFO [58]. The standard projection routines in ARC/INFO are then used to convert the data to the UTM coordinate system. In figure 4-15, a NEXRAD Stage III product is transformed from HRAP to UTM coordinates.

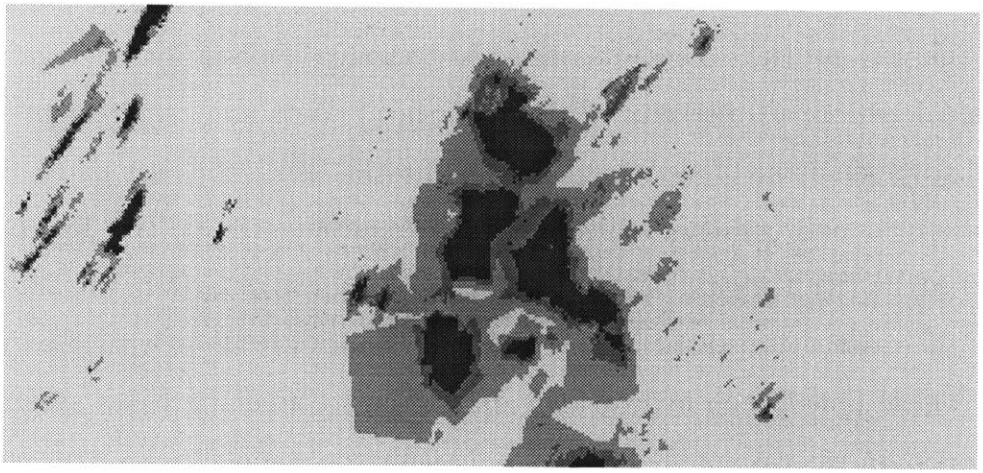
The data is clipped to fit the watershed as needed (Figure 4-16). An automated process to complete the above tasks has been created and can be found in Appendix B. This process also completes some of the initial data handling, including uncompression and moving files as needed, along with error detection processes.

4.2.2 Converting gridded rainfall to TINs

The last step involves translating the NEXRAD rainfall values from the grid to the TIN framework. As was mentioned in Chapter 3, All operations in tRibs are done using the voronoi area associated with each point in the TIN. Therefore, one needs to find the effective rainfall that falls upon each voronoi cell. Generally, due to the large size of each NEXRAD grid cell (approximately 16 km^2) the entire voronoi area associated with a pixel fits within a single grid cell. In these cases, the effective rainfall at that point in the TIN equals the rainfall at that grid cell. Occasionally, the voronoi area for a point in the TIN will overlap two or more grid cells. When this occurs, an areal average of the rainfall is used to find the effective rainfall for that point in the TIN. This entire process is completed using AML, and the code to do this is given in Appendix B. With the rainfall converted to the voronoi cells (Figure 4-17), the rainfall data is exported from ARC/INFO into the proper format for use in tRIBS.



A



B

Figure 4-15: Converting from HRAP coordinates (A) to UTM Coordinates (B)

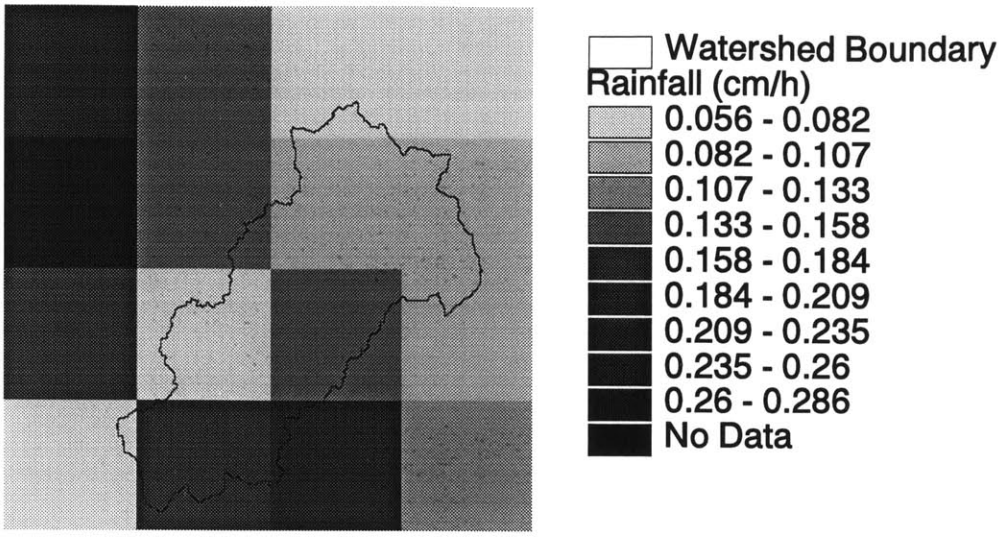


Figure 4-16: NEXRAD Rainfall clipped to the Peacheater Creek Watershed for May 31st, 1996 at hour 16z

4.3 All Other Data

The river reach (stream network) and gage location data were obtained using the data provided through the EPA BASINS (Better Assessment Science Integrating Point and Nonpoint Sources) project. These datasets also included information on regarding the size and location of reservoirs in the watershed. With the requirement that basins with large amounts of storage be rejected for this work, this was particularly useful. The stream network data available in BASINS includes the USEPA Reach File Version 3.0 Alpha (RF3-Alpha) dataset [22]. RF3-Alpha uses data from the first two versions of the reach files created by the EPA (RF1 and RF2) along with the USGS 1:100,000 Digital Line Graph hydrography data to create a complex new system of river networks. In the case of tRIBS, this is actually too complex with data points often located less than 10m apart. Using this raw data would cause numerous sliver triangles with very small areas. Therefore, the RF3 was simplified using ARC/INFO. The actual location and shape of the stream channel will be slightly different than the simplified network (Figure 4-18).

Soils data from the State Soil Geographic (STATSGO) database was also collected for use in the distributed flood forecasting model. Generated from the State Survey

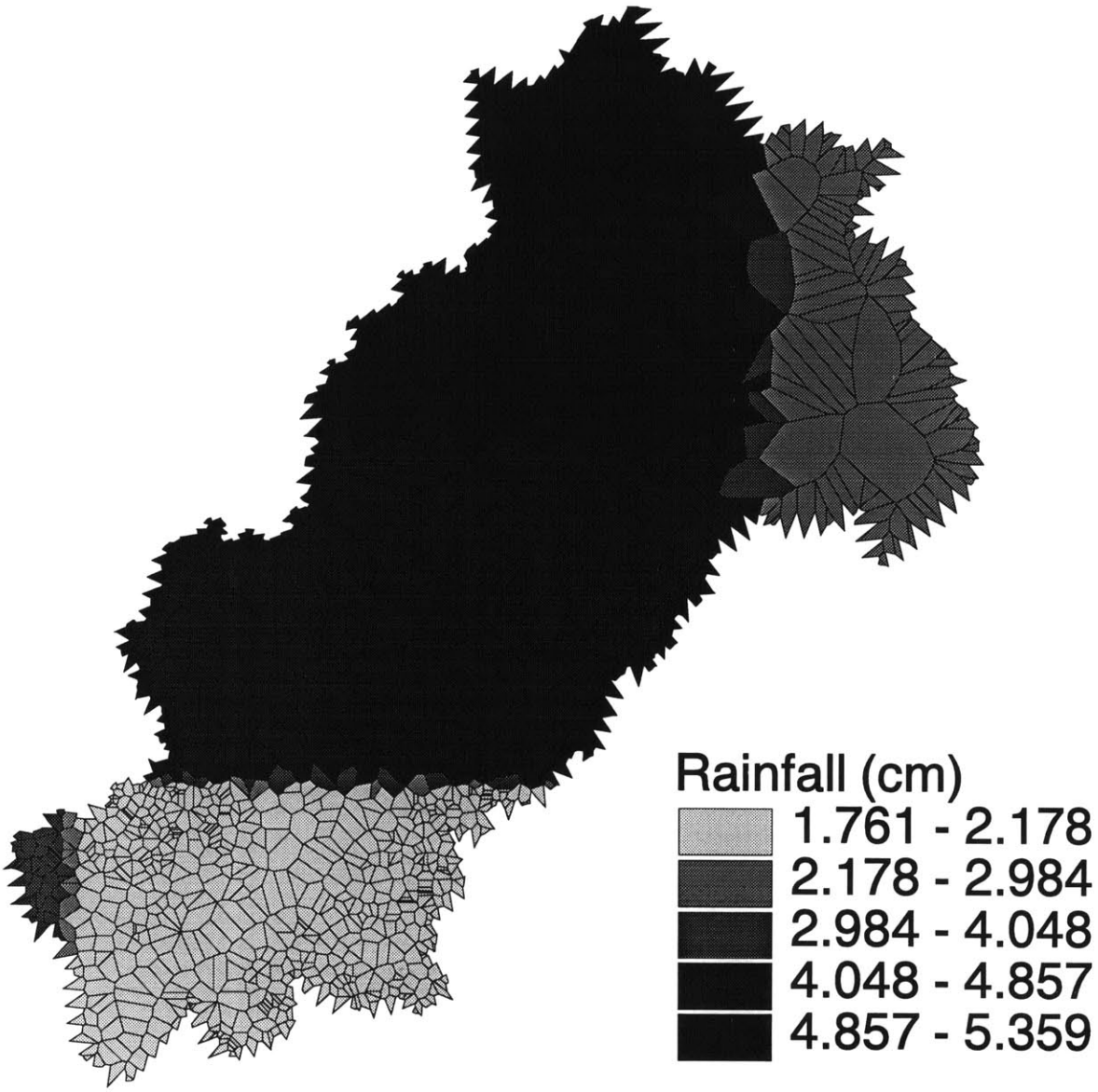


Figure 4-17: The effective rainfall over Peacheater Creek for September 26th, 1996 at 12z

Figure 4-18: An example of the difference between the original and a simplified stream networks in the Baron Fork watershed

Geographic (SSURGO) databases, geology, topography, vegetation, climate, and remotely sensed images, the STATSGO database provides a national soils map at the 1:250,000 map scale used by the USGS [13]. While other soil datasets may be attractive, the national coverage of STATSGO make it the dataset of choice here. If higher resolution of soils is needed in the future, many other regional datasets could be used in the place of STATSGO. The use of soils data in the tRIBS model is almost identical to that of the precipitation data. The voronoi cells of the watershed are overlain with the soils data, providing a soils class for the voronoi cell and its associated pixel. The only difference is that for precipitation one can have effective rainfall. There is no such thing as an effective soil class; therefore, when two soil classes intersect one voronoi area, the soil class with the largest area in that voronoi cell defines the soil class of that cell. Hourly streamflow data from the USGS for use in the calibration of the model and groundwater well data from a wide variety of sources is collected for use in defining the initial conditions of the distributed hydrologic model as well.

4.4 Publishing Distributed Data

As a final process in the creation of the distributed data set given here, all of the data has been posted on a public web site. This will allow others to avoid the difficult process of manipulating distributed data as all of the transformations have been completed. Also, all files are given in simple formats, so no special programs are required to access this data. These data sets can be found at:

http://platte.mit.edu/watdata/watdata.html

Chapter 5

Model Results

The tRIBS model is used in two model applications in this chapter. A simple hillslope model is used to test the dynamics and sensitivity of the model. The Peacheater Creek watershed in Oklahoma is modeled to test the capability of the model to properly predict streamflow under 'real world' conditions.

5.1 The Hillslope Model

As shown in Figure 5-1, a small triangular hillslope created from 75 points is used to test the dynamics and sensitivity of tRIBS. Each point is separated by 90 meters in the x and y direction, with a 45 m offset between rows to create the triangular pattern. These dimensions create 44 voronoi cells in the hillslope and each cell has a area of 8100 m^2. A water table depth for this simulation was computed using

$$N_{wt} = 250z \tag{5.1}$$

where N_{wt} is the groundwater table depth in mm, and z is surface elevation in meters. A cross-sectional profile of the surface and water table depths associated with the hillslope is given in Figure 5-2.

The parameters of the tRIBS model were assigned as shown in Table 5.1.

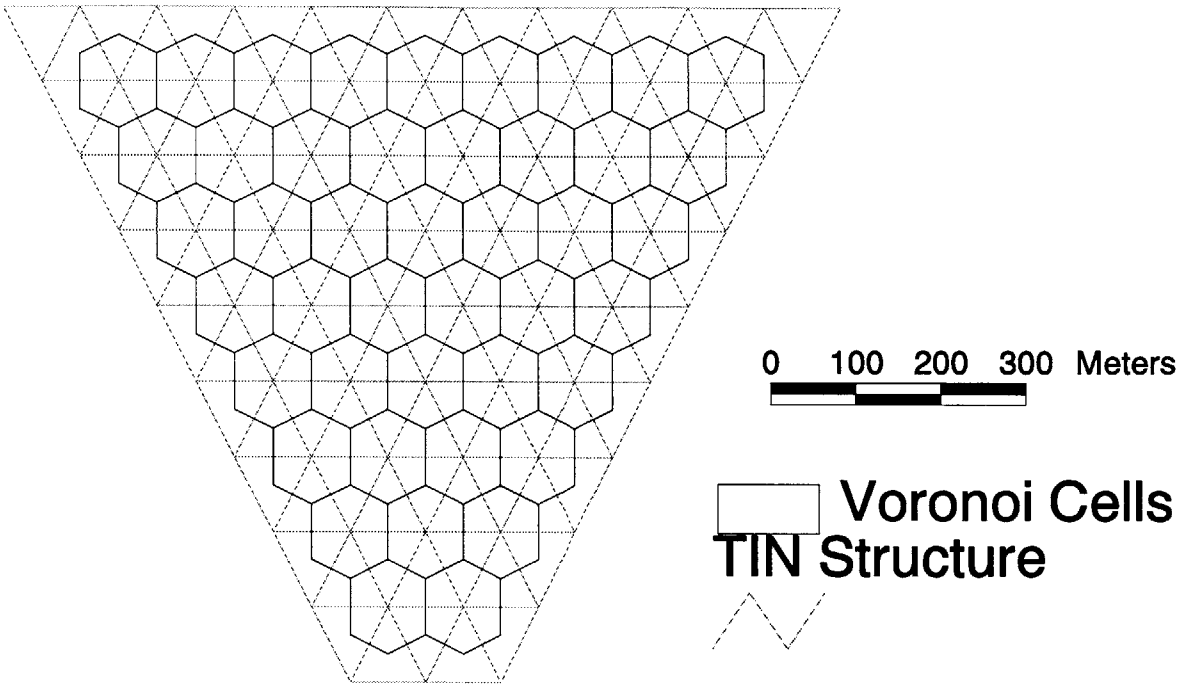


Figure 5-1: The hillslope used to test tRIBS

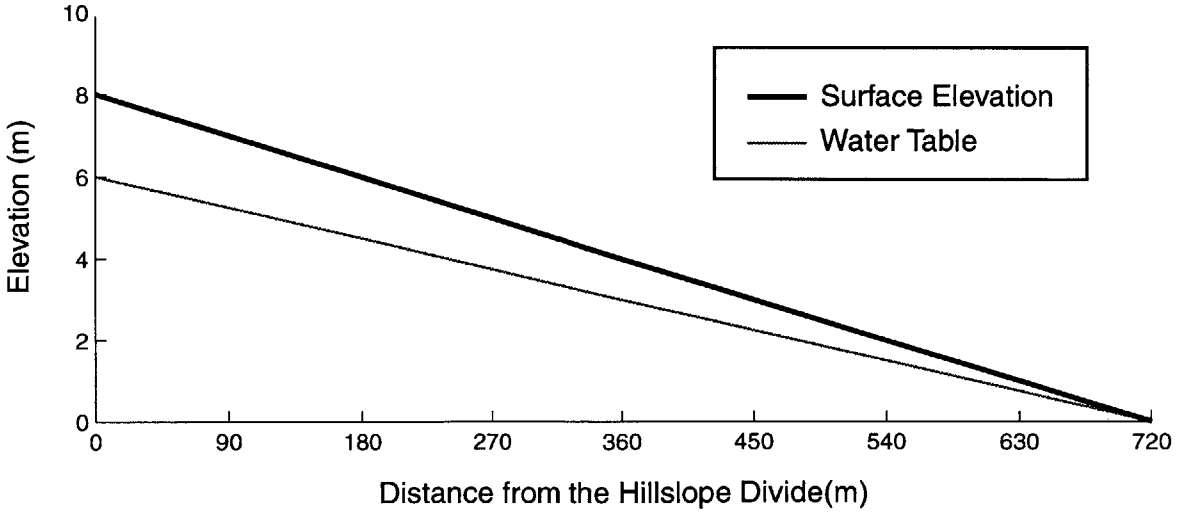


Figure 5-2: Cross-Sectional view of the hillslope used to test tRIBS

Table 5.1: Parameters used in the hillslope model

Parameter Name	Symbol	Parameter Value
Residual Moisture Content	θ_r	0.05
Saturated Moisture Content	θ_s	0.5
Saturated Hydraulic Conductivity (mm/hr)	K_{0n}	40
Decay of Hydraulic Conductivity (mm^{-1})	f	.001
Anisotropy Ratio	a_r	2
Pore Index	p_i	1.82
Porosity	n	0.45
Bubbling Pressure	Ψ_b	-800
Channel Velocity (m^3/sec)	C_v	0.75
Channel/Hillslope Velocity Ratio	K_v	40

5.1.1 Simulation One: A saturation event

Using a constant 5 mm/hr rainfall on the given hillslope, total saturation over the basin occurs in just under 40 hours. Figure 5-3 presents the evolution of these saturated areas in the hillslope during the first 6.25 hours of the storm. As seen in the figure, the first row of pixels closest to the outlet reach saturation after 0.75 hours. Saturation occurs for the first time at this step, so only a fraction of the total available rainfall generates runoff. At the next time step (hour 1), all of the rainfall falling on the saturated voronoi areas runs off. As time progresses, the saturated area of the hillslope increases as row 2 reaches saturation (hour 3) and row 3 follows (hour 6). Although it is not shown, this process continues until all pixels are saturated after 37.5 hours of a constant 5 mm/hr rainfall event.

The movement of the top and wetting fronts are illustrated in Figure 5-4. When the first row of voronoi cells become saturated at 0.75 hours, all other pixels are still unsaturated. As seen in the first panel of Figure 5-4, the wetting front and top front are equal throughout the hillslope, moving downward as an unsaturated wedge of moisture. The only change occurs in the first row where the pixel is fully saturated and the top and wetting fronts collapse and disappear at the surface. At 3 hours, the second row of voronoi cells has reached saturation. This is visible in Figure 5-

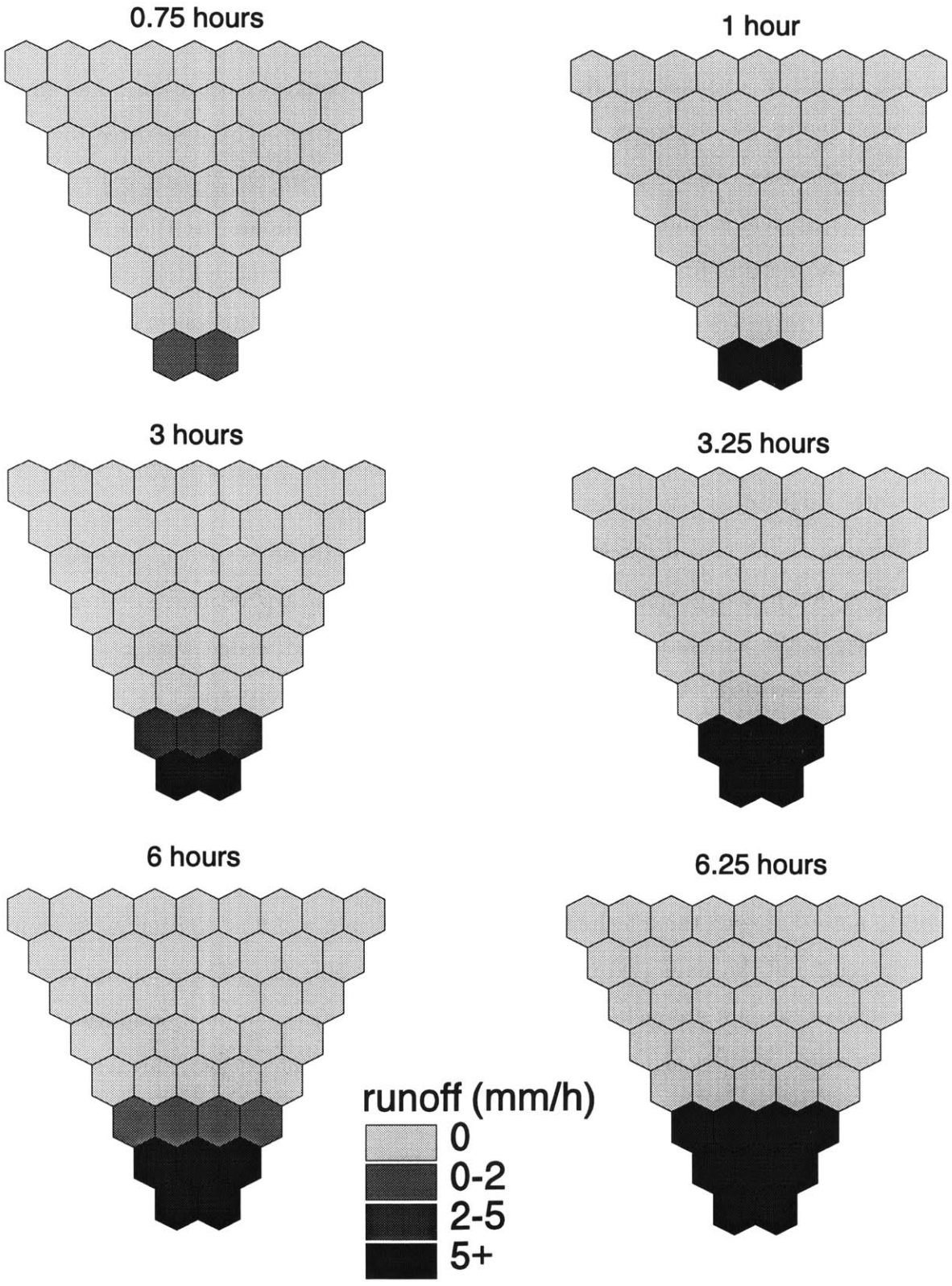


Figure 5-3: Runoff from the hillslope under a constant 5mm/hour rainfall event

4, panel 2 with the extension of the region where top and wetting fronts coincide at the surface. Also shown in panel 2 is the development of a separate top front in the third row of pixels. This row has developed perched saturation. The final panel of Figure 5-4 illustrates the front locations at hour 6. The first three rows of voronoi cells have reached saturation (wetting and top front at the surface) and the fifth row of pixels is undergoing perched saturation conditions. At the fourth row of pixels the wetting front of a perched saturated pixel has reached the water table. Instead of creating a new wetting front at the surface when this occurs, the water table is raised to incorporate the incoming water. The effect of this evolution of the groundwater (shown in Figure 5-5) is due to the newly added "storm evolution" pixel state described in Chapter 3. The water table profile shown is flattened at row 4 as the water table adjusts to incorporate the saturated region.

Looking at the row of pixels closest to the hillslope divide, a plot of wetting and top front depth versus time (Figure 5-6) shows numerous pixel states. From the beginning of the simulation until hour 14, an unsaturated wedge without perched saturation is formed. At hour 14, perched saturation develops and this pixel state continues until hour 19 when "storm evolution" begins and the groundwater rises as it incorporates the fully saturated region. The final pixel state of full saturation is reached after 37.5 hours.

The hydrograph associated with the simulation is given as Figure 5-7. It is important to note that even though the hydrograph shows values of flow continuing to hour 70, the length of the simulated rainfall event was only 50 hours. The 20 hours added to the end of the plot are included to account for runoff at the upper boundary of the hillslope which requires significant travel time to drain to the outlet. This additional hours of the hydrograph are important as they are needed to complete mass balance in the system. Mass conservation in the tRIBS model is checked using the following

$$M_i = M_o \pm M_s \tag{5.2}$$

where M_i, M_s, M_o are the mass of the inflow, storage, and outflow over the

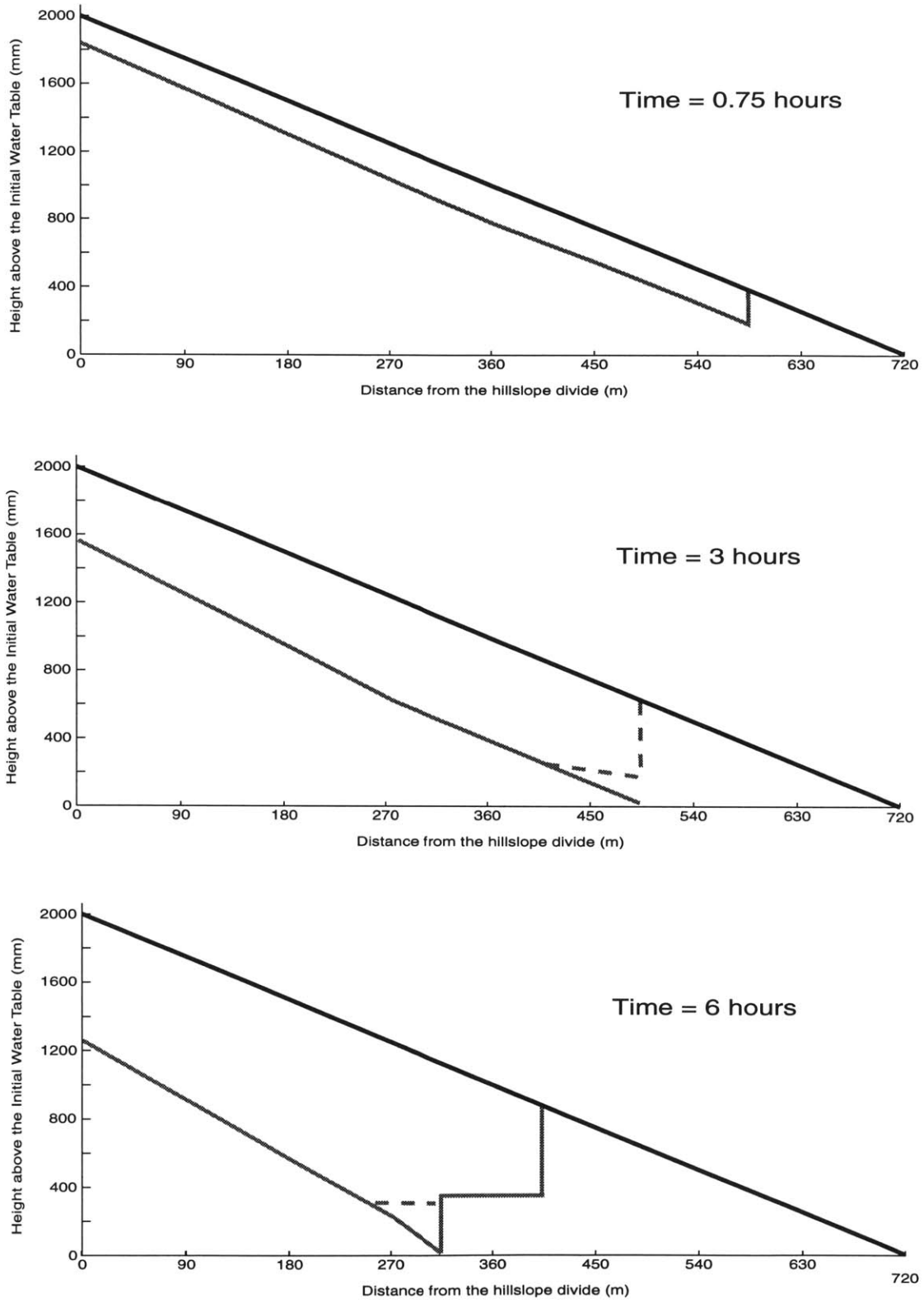


Figure 5-4: Development of a wetting and top fronts under a 5mm/hour rainfall event for three time periods. The surface is given by the solid black line, wetting front by a solid gray line and the dashed line represents the top front.

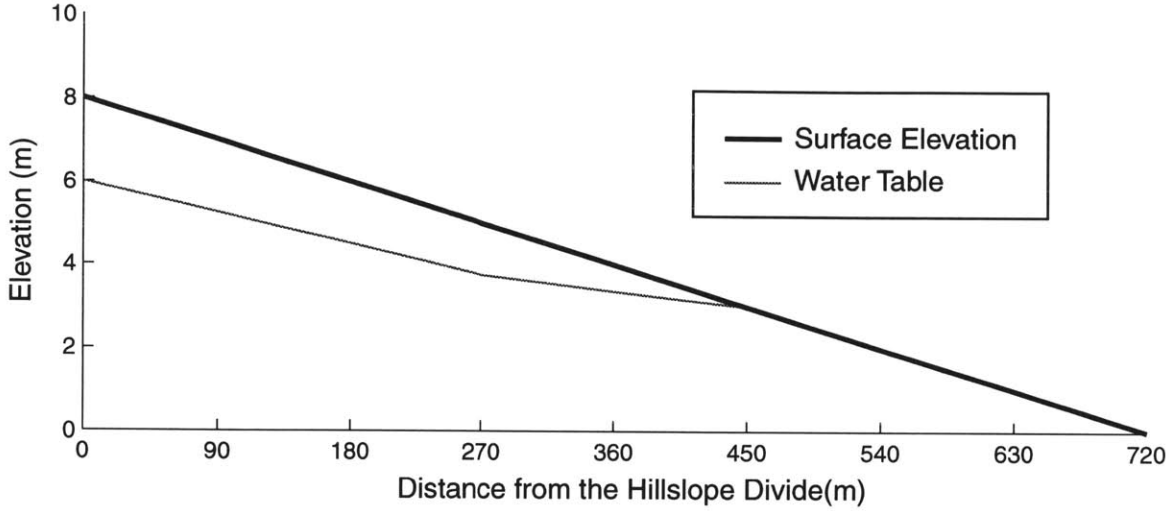


Figure 5-5: The hillslope surface elevation and water table depth after 6 hours of rainfall (intensity = 5mm/h)

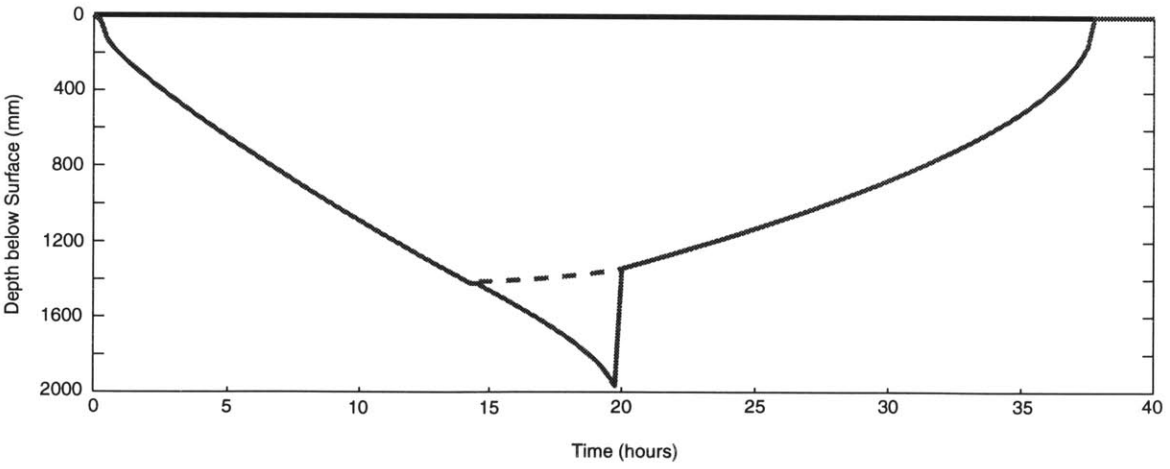


Figure 5-6: Wetting front (solid gray line) and top front (dashed gray line) depth for a pixel closest to the hillslope divide versus time

watershed area. Assuming a constant density of water and no inflows other than rainfall

$$V_r = V_o \pm V_s \tag{5.3}$$

where V_r, V_o, and V_s are the rainfall, outflow and storage volumes in the system. In the tRIBS model storage in each pixel under complete saturation can be given by

$$S_i = \theta_s N_{wt} - M_i \tag{5.4}$$

where S_i is the volume of water per pixel area (in mm) stored during the simulation given earlier. $\theta_s N_{wt}$ is the total storage capacity of the unsaturated zone in each pixel and M_i is the initial amount of water found in the unsaturated zone at the beginning of the simulation. Combining this with expressions for rainfall and outflow volumes leads to

$$T \times R \times A \times n = \sum_{i=0}^{\tau} Q_i dt + \sum_{\in basin} S_i A_i \tag{5.5}$$

where T equals the length of the simulated rainfall, R is rainfall intensity, A is pixel area, n is the number of pixels, τ is the entire time period of the hydrograph, and Q is the basin outflow.

Calculating this balance for the simulation presented

$$89,100 = 52,181 + 36,554 \tag{5.6}$$

less than 0.5% of the mass is lost.

5.1.2 Simulation Two: Rainfall and interstorm conditions

A second test of the hillslope model begins with a 17 hour rainfall event (rainfall intensity = 5mm/h) followed by a 33 hour period of no rainfall. This simulation uses the parameters given earlier (Table 5.1), but a new deeper water table is used

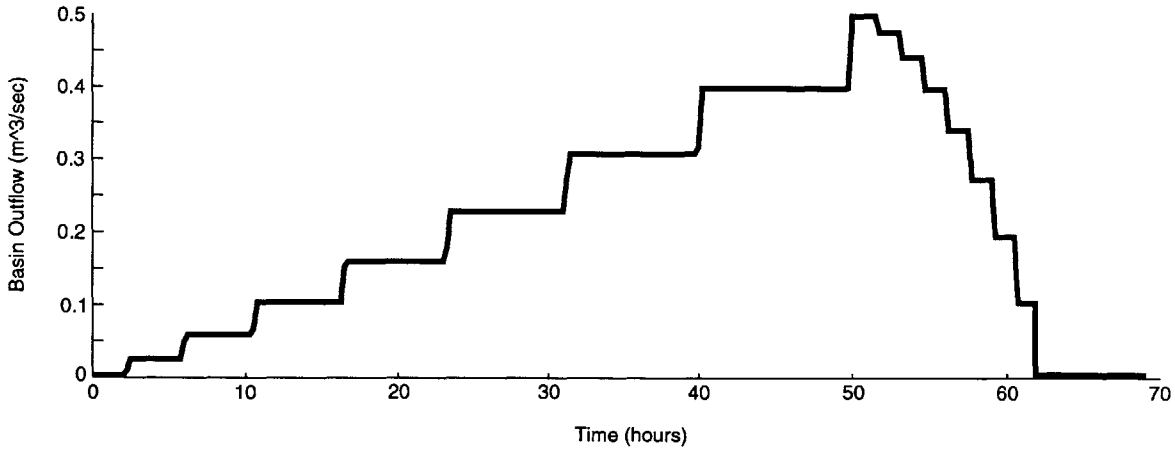


Figure 5-7: Hydrograph for the hillslope simulation

(Figure 5-8) to allow for longer times to saturation within the hillslope simulation. As shown in Figure 5-9, only the first two rows of pixels are saturated in this system before the interstorm period begins. This deep water table also allows for the further development of the perched saturated and "storm evolution" pixel states. For a typical voronoi cell in the fourth row of the hillslope, front evolution is as shown in Figure 5-10. In this simulation, a coincident wetting and top front descend under constant rainfall for the first 13.5 hours of the simulation. At this point, perched saturation occurs with an ascending top front and descending wetting front. With the end of the rainfall event at hour 17, the top front begins to descend due to the lack of incoming moisture. As the wetting front intersects the water table (hour 19.75), the moisture in the pixel is redistributed and the water table rises. With no additional external moisture input and only limited lateral flows, this new condition remains stable for the rest of the simulation. With saturation only in the first two rows of pixels, runoff is quite small for this simulation, and the hydrograph (Figure 5-11) shows this quite well.

5.1.3 Simulation Three: Two storm events

Using the parameters (Table 5.1) and water table depth (Figure 5-8) from the second simulation, 17 hours of rainfall are applied, followed by 9 hours without rain, and ending with 24 more hours of rainfall. The wetting and top front evolution (Figure

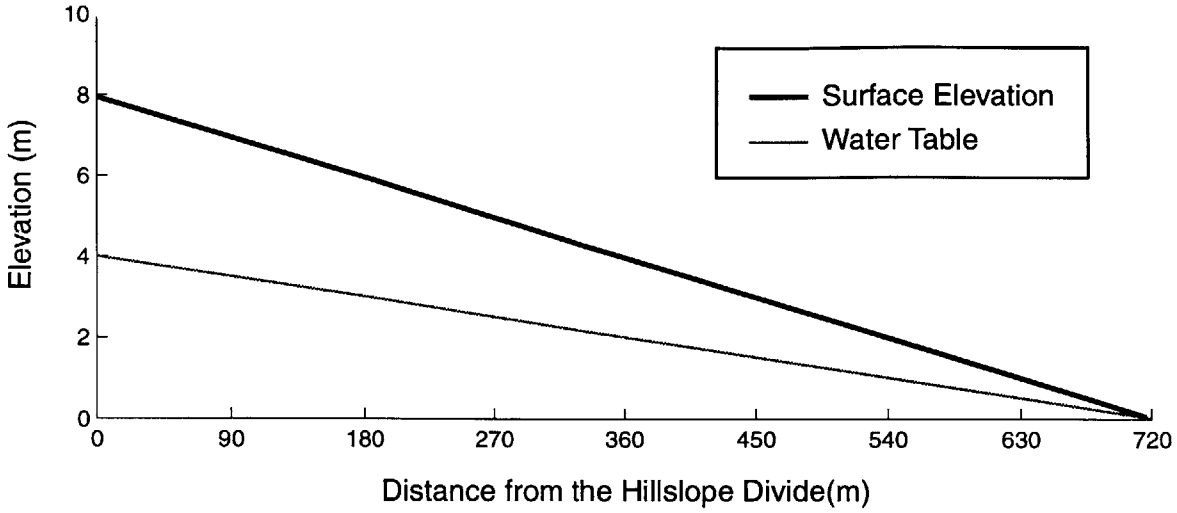


Figure 5-8: Cross-Sectional view of the hillslope used in the second tRIBS test simulation

5-12) are the same as that calculated in simulation two until hour 26. With the addition of a second rainfall event at this time, the water table begins ascending to the surface, reaching saturation at hour 47.25. The hydrograph associated with this simulation (Figure 5-13) shows the two peaks as expected from these two separate rainfall events. This figure also shows a spurious peak in the hydrograph after hour 50. The peak is caused by the linear routing scheme used in tRIBS in combination with the small number of pixels used in the hillslope simulation. As constructed, the simulated hillslope has only 44 voronoi cells. With this small number of voronoi cells, a linear routing scheme may show errant spikes as seen here due to the coarse resolution of the system. As simulations move towards basin scale systems with thousands of points, this issue of errant spikes becomes insignificant.

5.1.4 Model sensitivity

The model sensitivity to f (Figure 5-14), A_r (Figure 5-16), and C_v (Figure 5-17) were studied using the first hillslope simulation. The model is quite sensitive to f. In tRIBS, f is one of the most significant parameters in defining the soil moisture storage in the unsaturated zone. Typically, one would expect that as f decreases, the rate of change of hydraulic conductivity with depth in the soil column will decrease

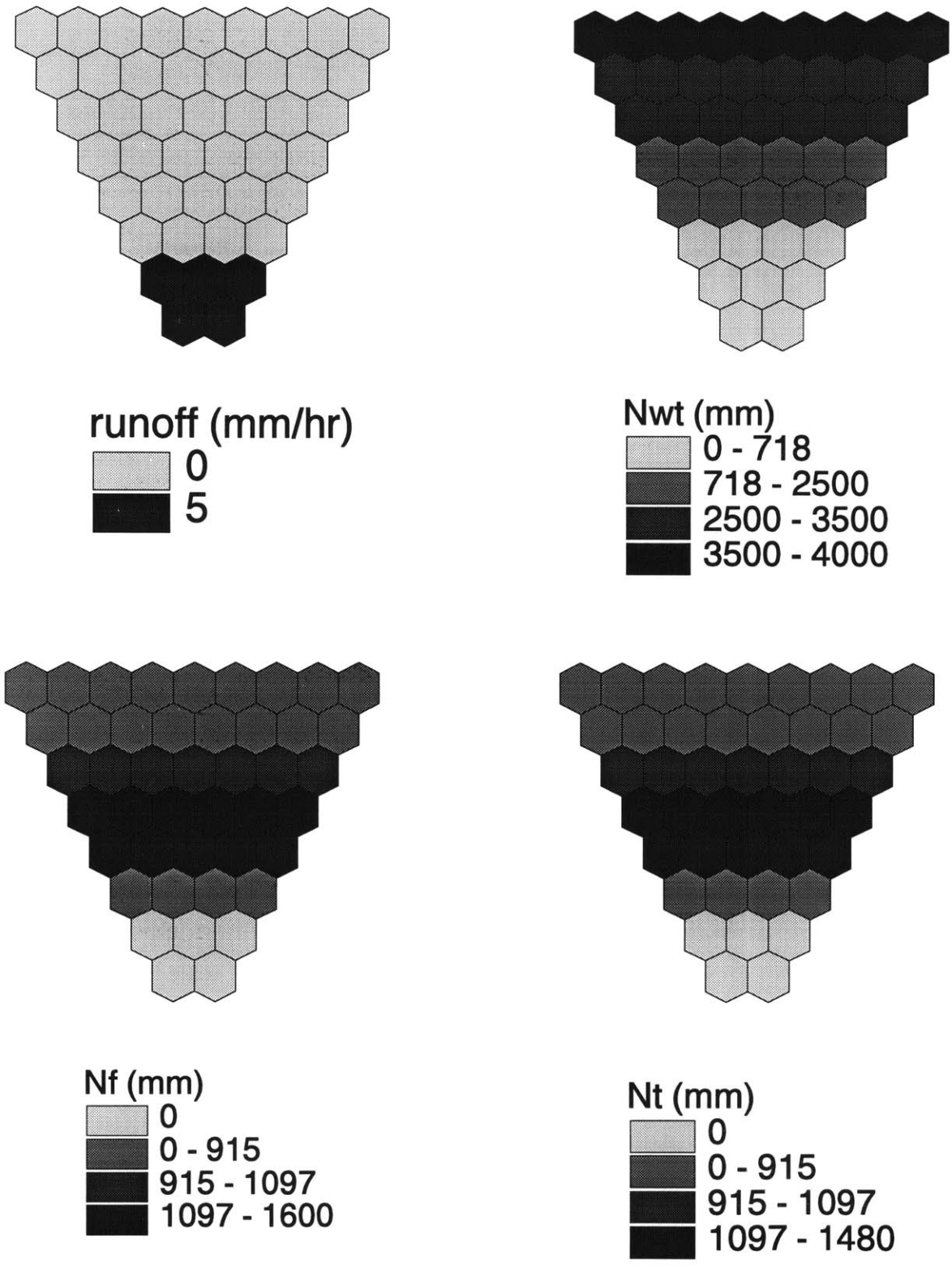


Figure 5-9: The status of various pixel variables at 16.75 hours for the second tRIBS hillslope test

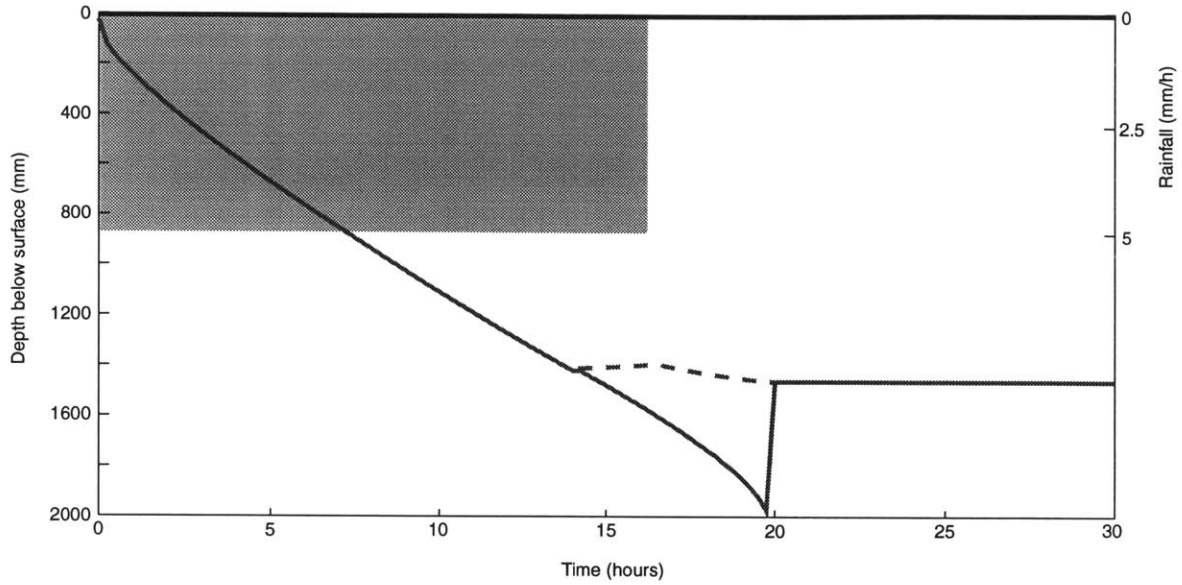


Figure 5-10: Wetting and top front evolution for a pixel under the second tRIBS hillslope test

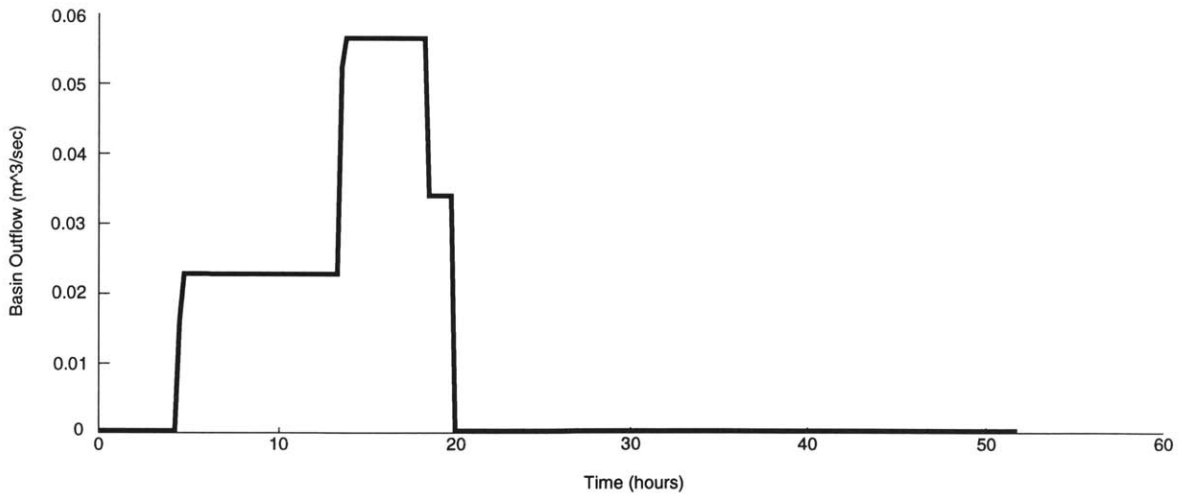


Figure 5-11: Hydrograph for the second hillslope simulation

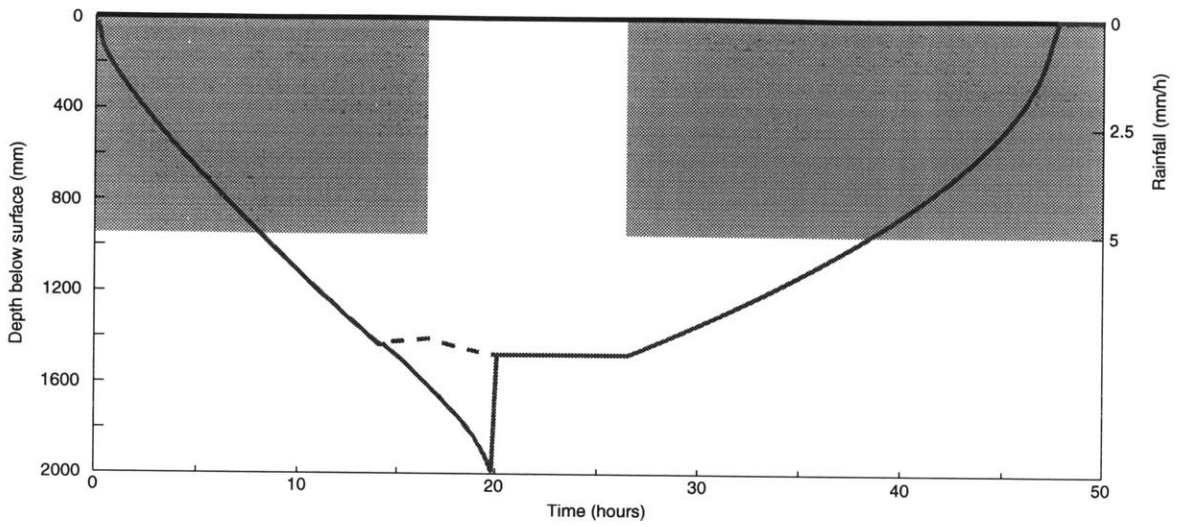


Figure 5-12: Wetting and top front evolution for a pixel under the third tRIBS hillslope test

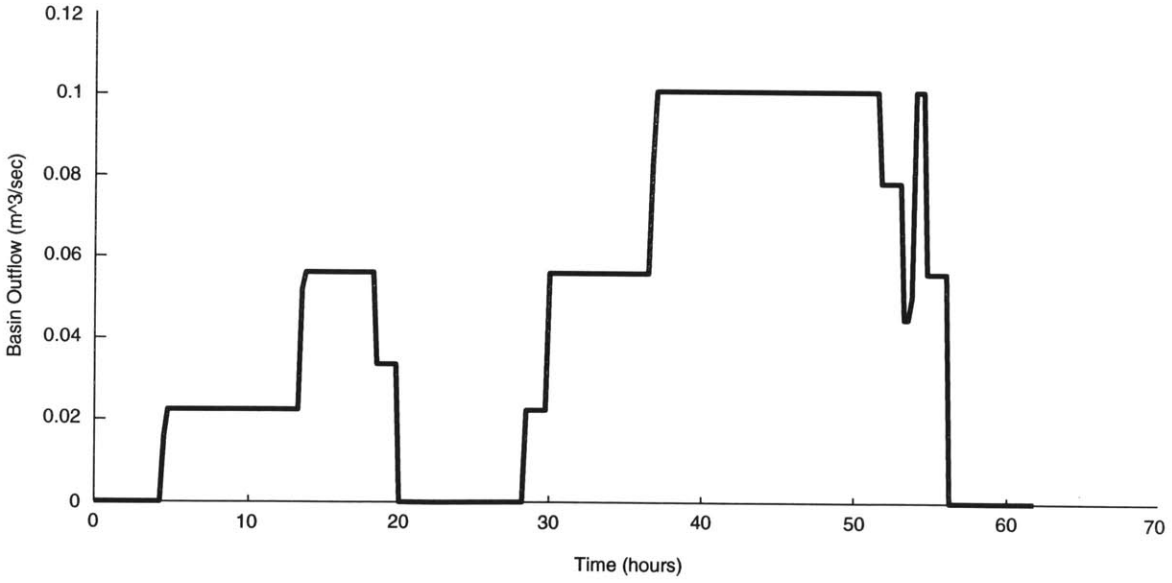


Figure 5-13: Hydrograph for the third hillslope simulation

as well. This in turn leads to decreased runoff as the soil column can transmit water more efficiently to lower depths in the soil column. This is not true in the version of tRIBS used here. As f decreases, soil moisture storage decreases, leading to faster saturation in the pixel under similar rainfall events. This behavior is caused by the method used to initialize the moisture content of the unsaturated zone in the tRIBS model. Due to assumptions in the tRIBS model, initial moisture content, M_i, is formulated as

$$M_i = \left(1 - e^{\frac{-fNwt}{\epsilon}}\right)(\theta_s - \theta_r)\left(\frac{\epsilon}{f}\right) + \theta_r Nwt. \tag{5.7}$$

With this initialization, the value of f has a significant role. As shown in Figure 5-15, a small f value leaves very little unsaturated area in the soil column for incoming storm moisture. This area is quickly filled during a storm event and runoff occurs. This behavior dominates the effect of f on tRIBS and creates this unusual response which has been modified by Valeri Ivanov [39] in a newer version of tRIBS. In that version, initialization of the unsaturated zone assumes a zero flux equilibrium condition rather than the constant flux assumption that leads to Equation 5.7 and results in the aberrant dependence of initial moisture state on f.

Varying anisotropy has an effect on tRIBS, but it is not as obvious. Increasing anisotropy will increase lateral flow in the system. The effect of this increased lateral flow can be seen in the hydrograph, but only along the rising limb. With increased lateral flow, saturation will occur slightly earlier than seen with less significant lateral flow. Sensitivity to channel velocity is shown in the timing of the hydrograph. As channel velocity increases peaks tend to occur earlier. The magnitude of the hydrograph peak may also be altered due to changes in the aggregation of flow with differing velocities.

5.2 Peacheater Creek

The Peacheater Creek watershed located in eastern Oklanhoma was chosen as the test basin for tRIBS. This $65km^2$ watershed has been used previously by the National

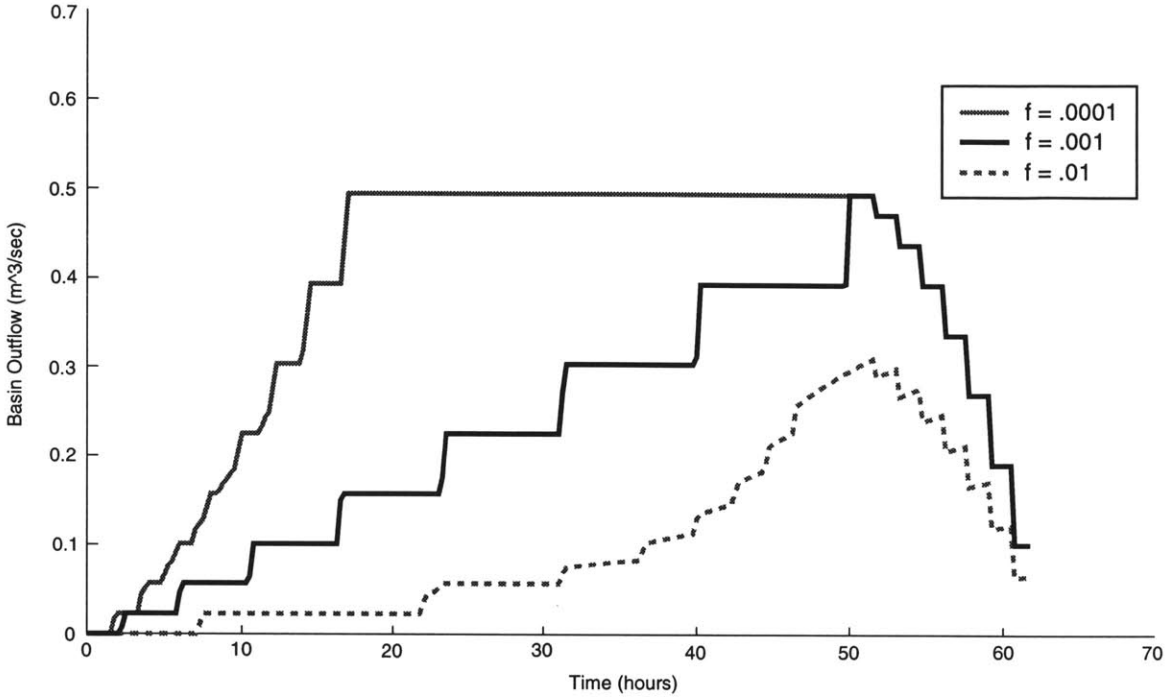


Figure 5-14: Hydrographs for varying f (mm^{-1}) values

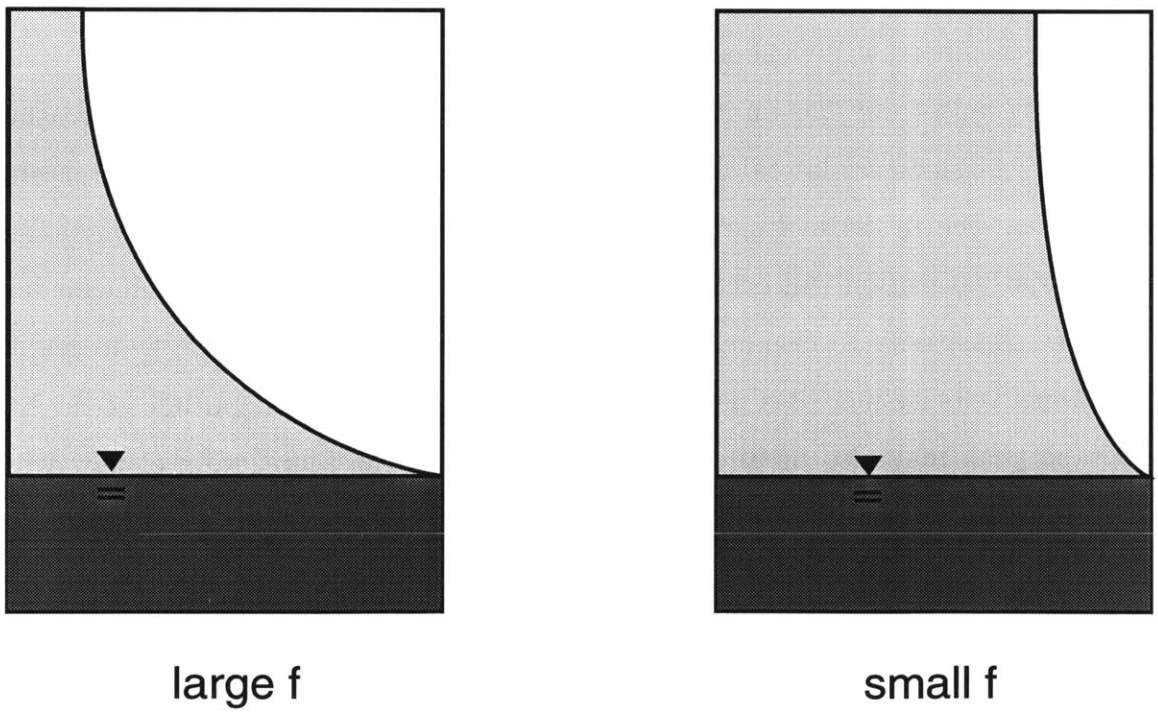


Figure 5-15: The initial mositure content of a tRIBS pixel as defined by f

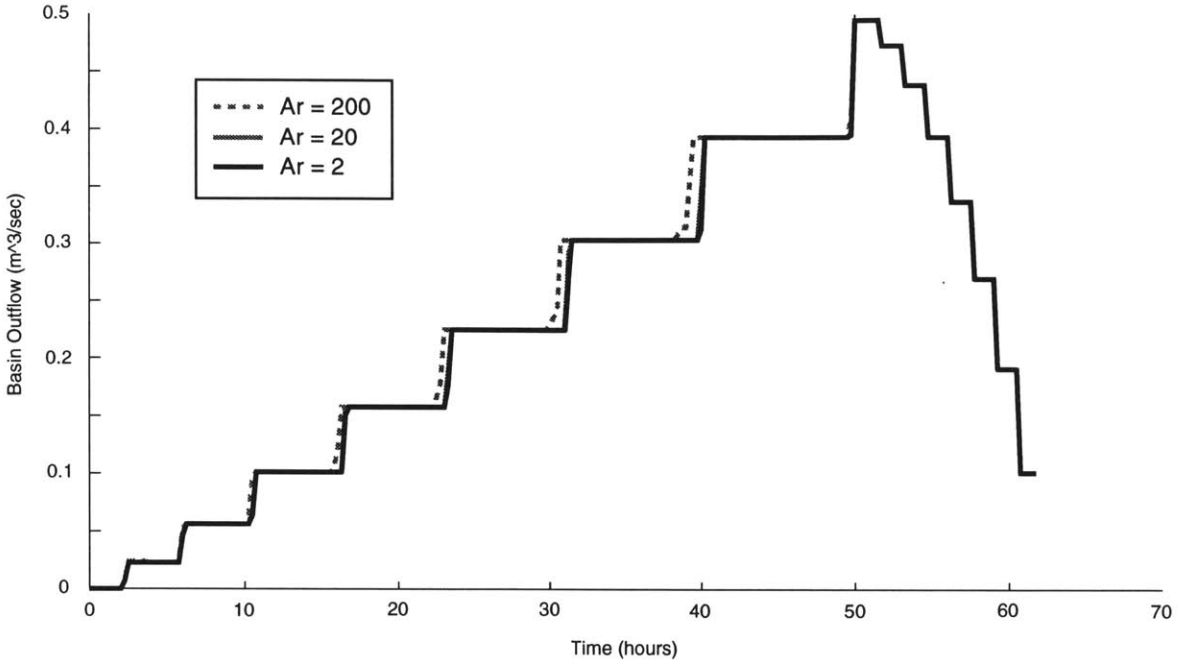


Figure 5-16: Hydrographs for varying anisotropy ratios (Ar)

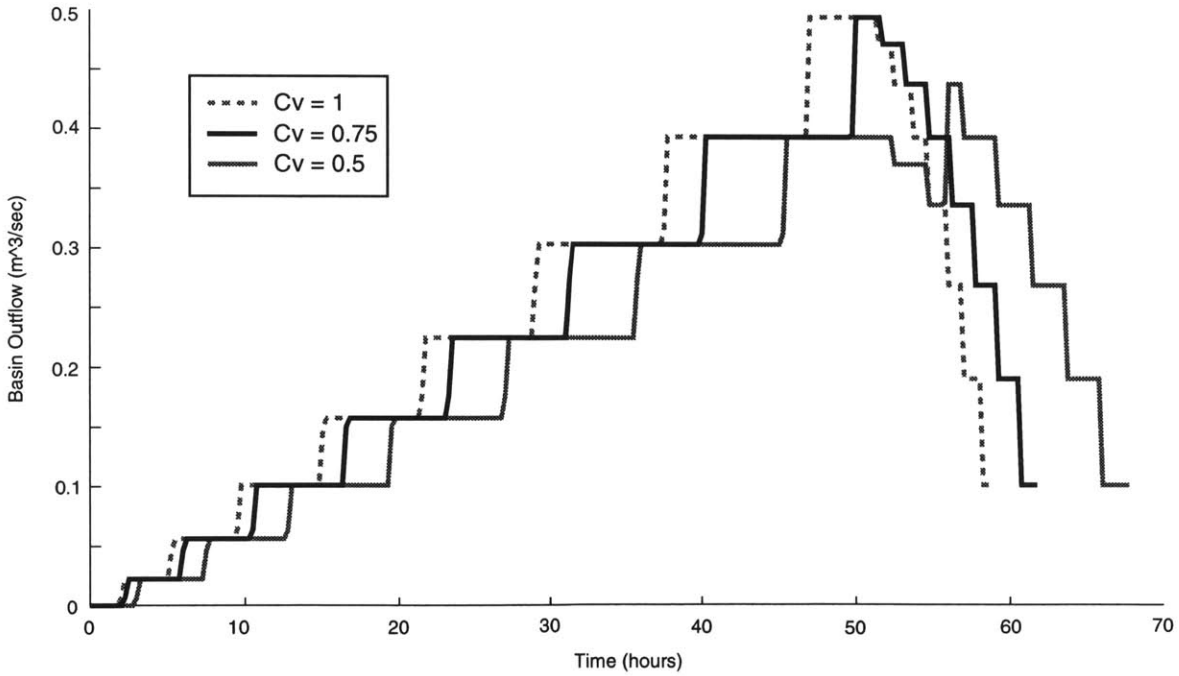


Figure 5-17: Hydrographs for varying hillslope velocity ratios (Cv)

Weather Service for their own testing of rainfall/runoff models. The topography of the watershed, given in Figure 5-18, shows a moderately sloping watershed with a significant plateau at the headwaters of the basin. The general direction of flow in the basin is from northeast to southwest. The watershed is unurbanized and contains almost no water storage, fitting the criteria given for a model test basin in Chapter 4.

The initial groundwater table is set using the method developed by Ivanov [39]. This procedure described in Chapter 3 is first run on the original gridded DEM of Peacheater Creek. This gridded water table is then input to the voronoi cells of the tRIBS model using the method for incorporating gridded rainfall shown in Chapter 4. The initial groundwater table used in this simulation can be found in Figure 5-19.

A calibration event from September 24th to September 30th, 1996 was selected for this basin due to some unique characteristics of this time period. Before September 24th, there are over three months without basin outflow in the Peacheater Creek watershed (Figure 5-20) which allows for the use of initially dry conditions in the tRIBS model beginning the simulation. Starting on the 24th of September, two storms occur within a six day time period. One small event with a maximum rainfall intensity of just over 20 mm/h precedes a larger rainfall event with a maximum rainfall intensity of over 50 mm/h. (Figure 5-21). These two storm events will allow for a good test of both storm and interstorm conditions in the tRIBS model.

In the calibration of tRIBS, an attempt to fit both runoff volume and travel time were completed. By adjusting K_{0n}, f, a_r, and C_v, a good fit of the actual streamflow during this time period was determined. This result shown in Figure 5-22 was modeled using the parameters given in Table 5.2. All of these parameters fit within the reasonable ranges expected when examining the physical characteristics of the Peacheater Creek watershed. As an additional check on the calibration of the model, evolution of saturated areas for various simulation times were plotted to see how individual pixels responded to the storm event. (Figure 5-23) As expected, more voronoi areas along the streams become saturated as rainfall events occur in the simulation.

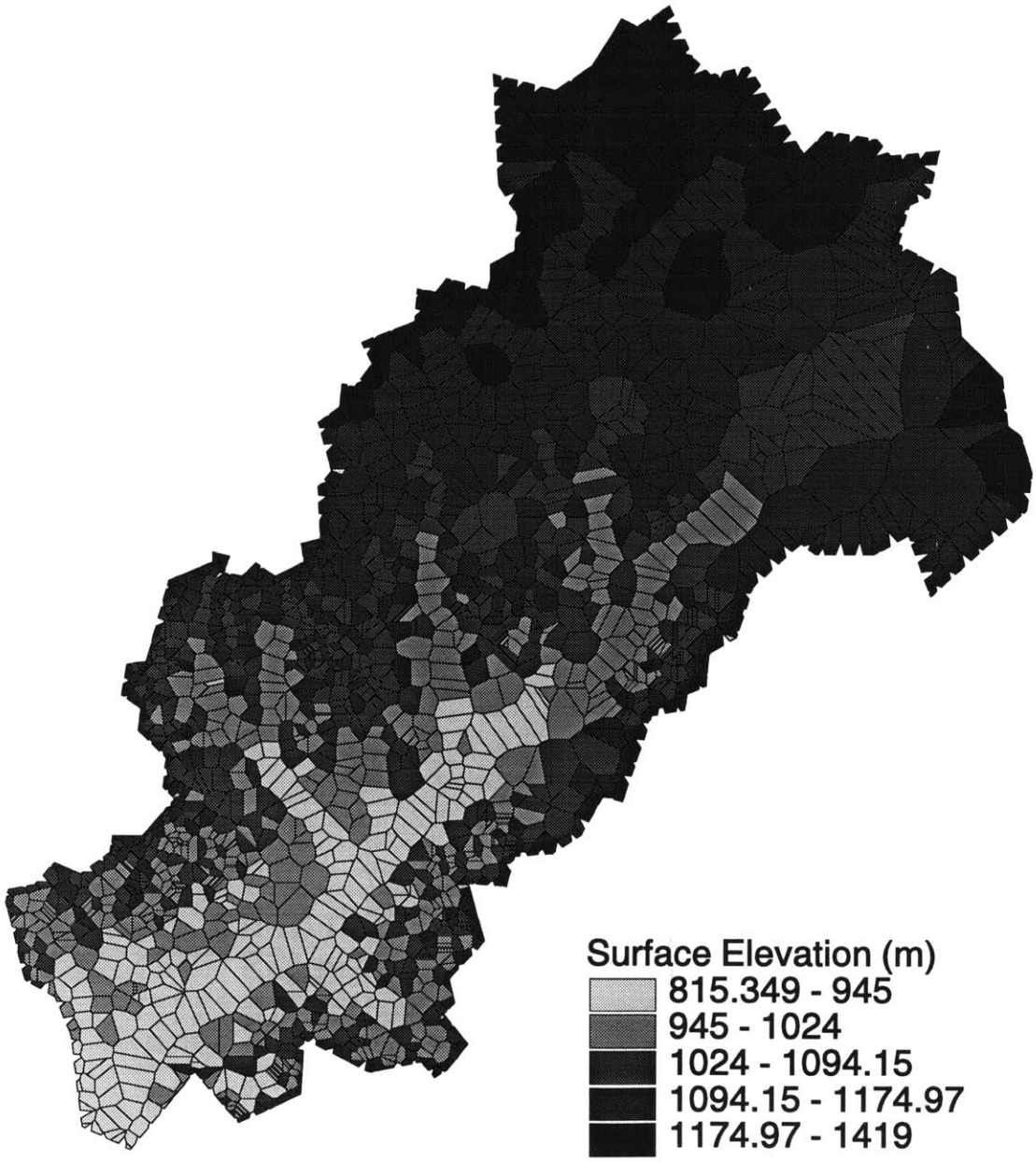


Figure 5-18: Surface Elevation of the Peacheater Creek watershed

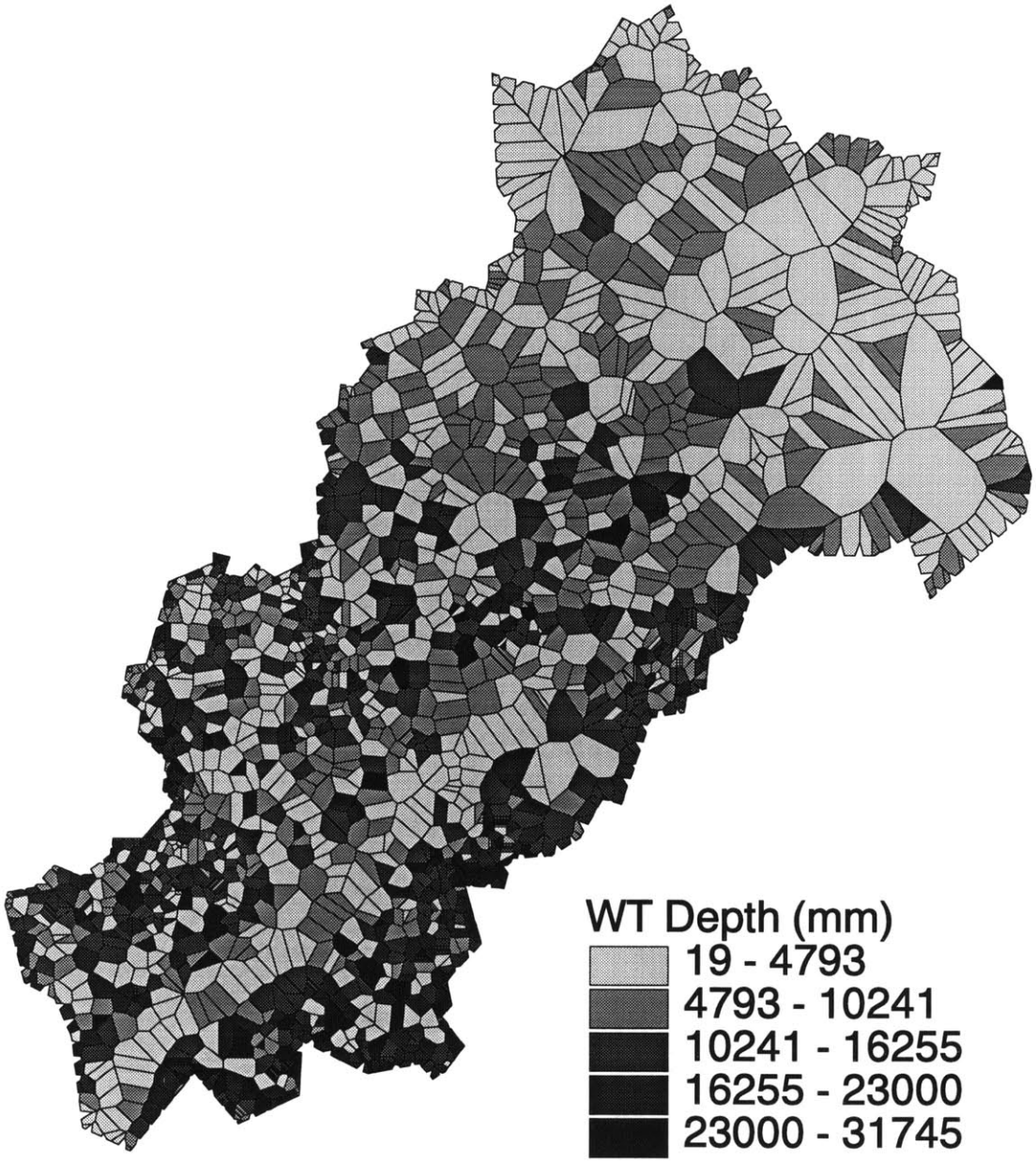


Figure 5-19: Initial Groundwater depth in the Peacheater Creek watershed

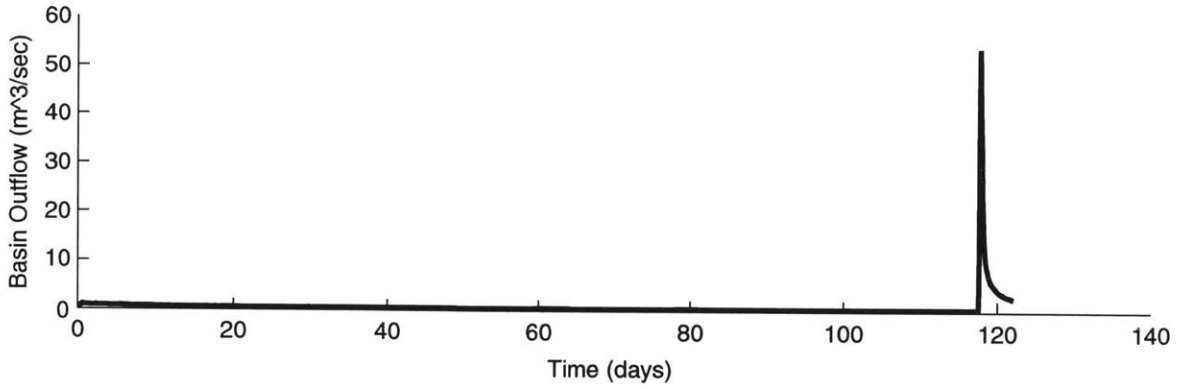


Figure 5-20: Basin outflow for the Peacheater Creek watershed, June through September 1996

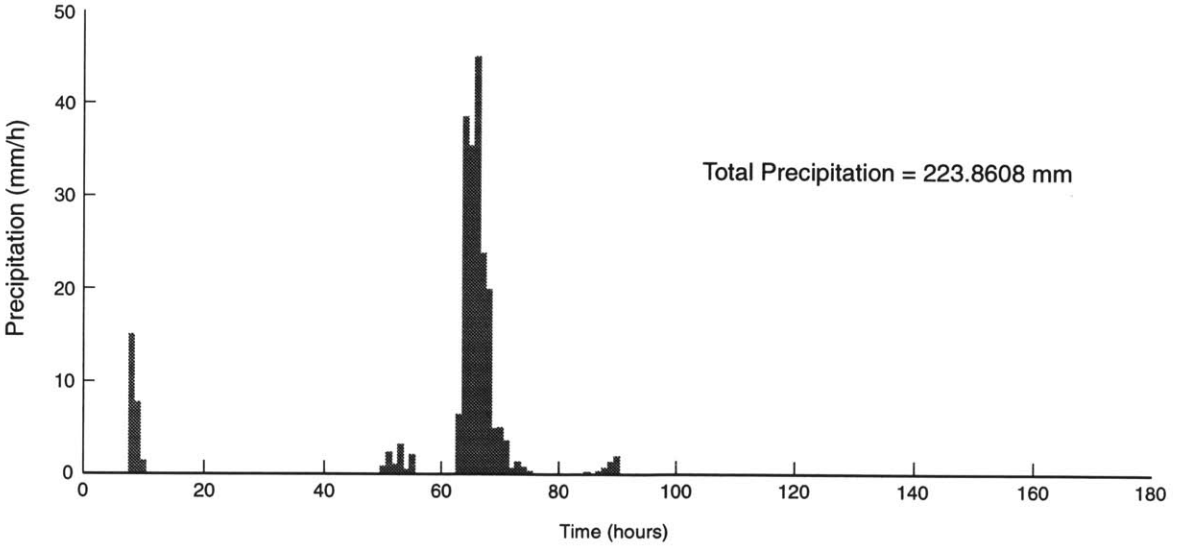


Figure 5-21: Areal average precipitation over the Peacheater Creek watershed for September 24th–30th, 1996

Table 5.2: Parameters used in the Peacheater Creek simulation

Parameter Name	Symbol	Parameter Value
Residual Moisture Content	θ_r	0.05
Saturated Moisture Content	θ_s	0.5
Saturated Hydraulic Conductivity (mm/hr)	K_{0n}	48
Decay of Hydraulic Conductivity (mm^{-1})	f	.0025
Anisotropy Ratio	a_r	500
Pore Index	p_i	1.82
Porosity	n	0.45
Bubbling Pressure	Ψ_b	-800
Channel Velocity (m^3/sec)	C_v	0.8333
Channel/Hillslope Velocity Ratio	K_v	40

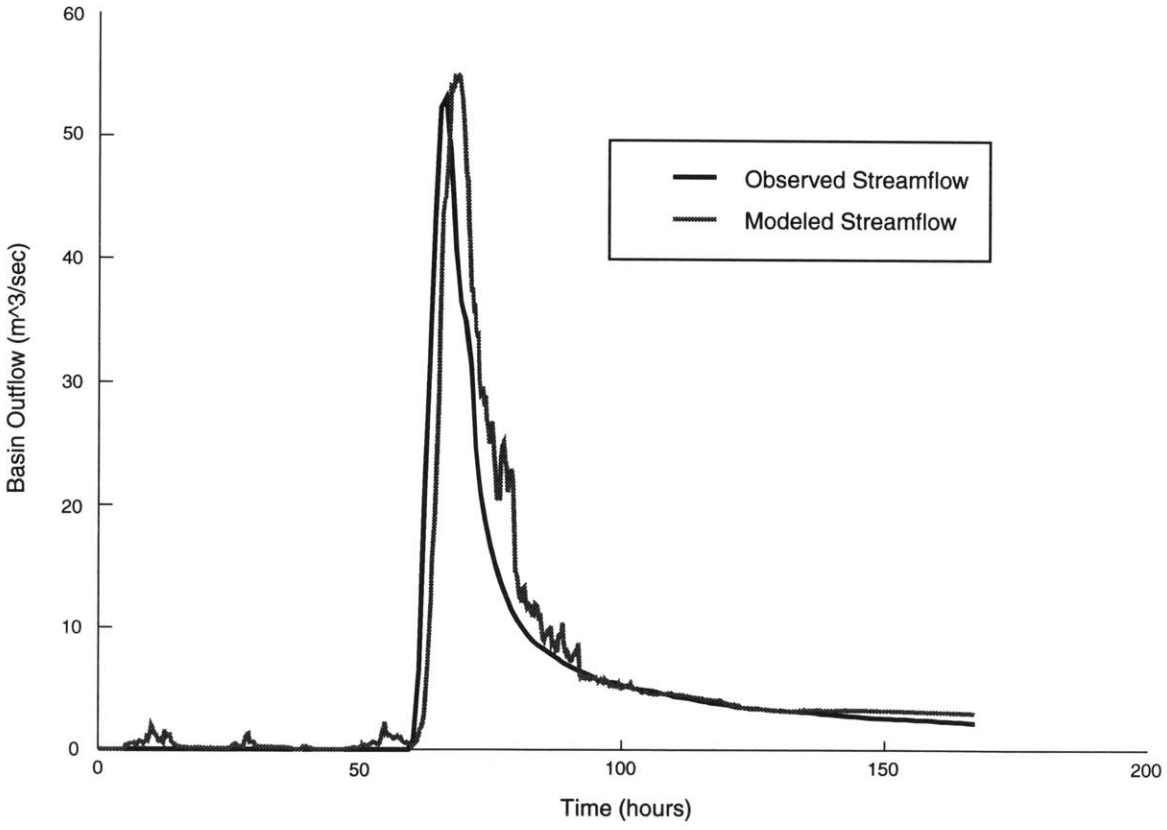


Figure 5-22: Observed and modeled streamflow for the outlet of the Peacheater Creek watershed, September 24–30th, 1996

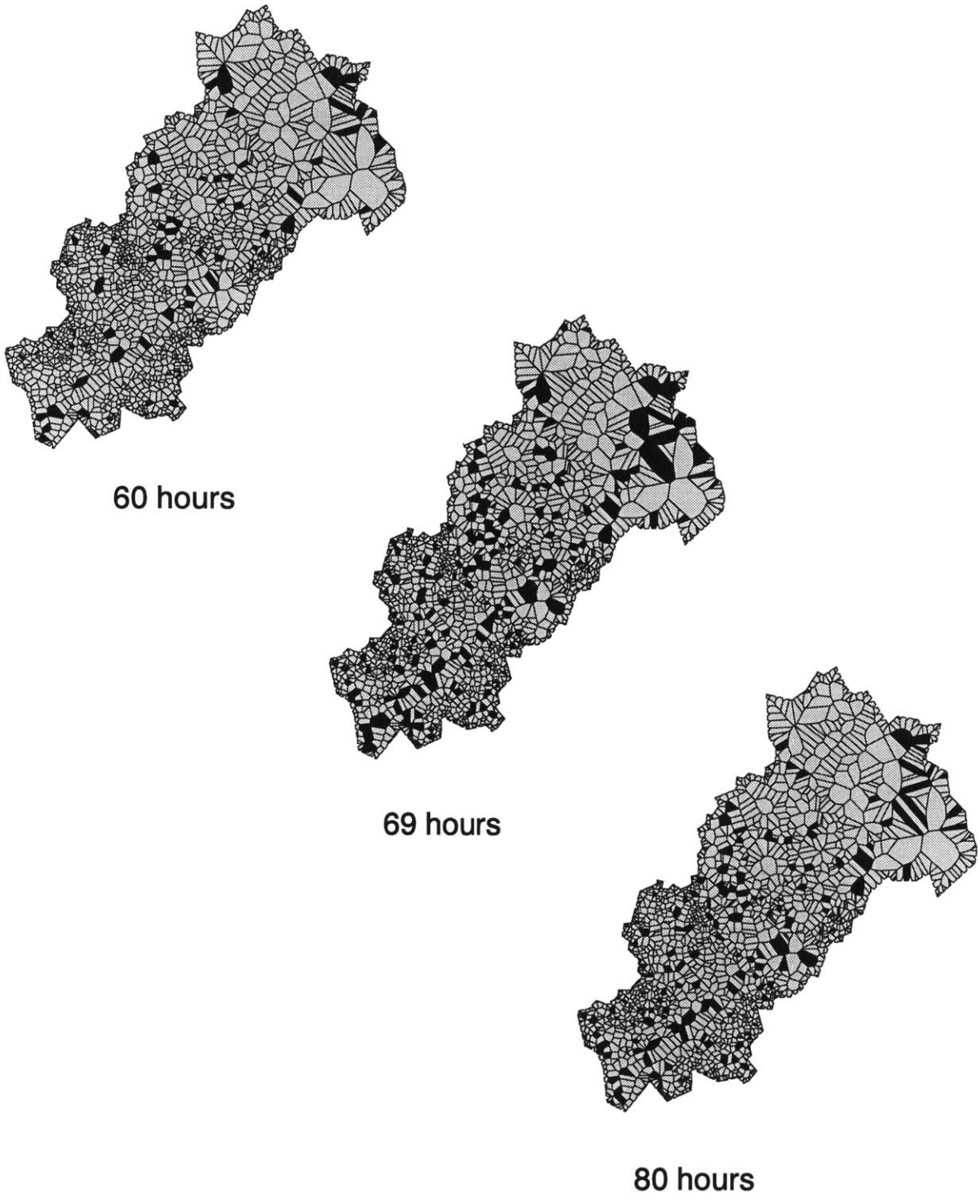


Figure 5-23: Evolution of saturated areas (in black) in the Peacheater Creek watershed

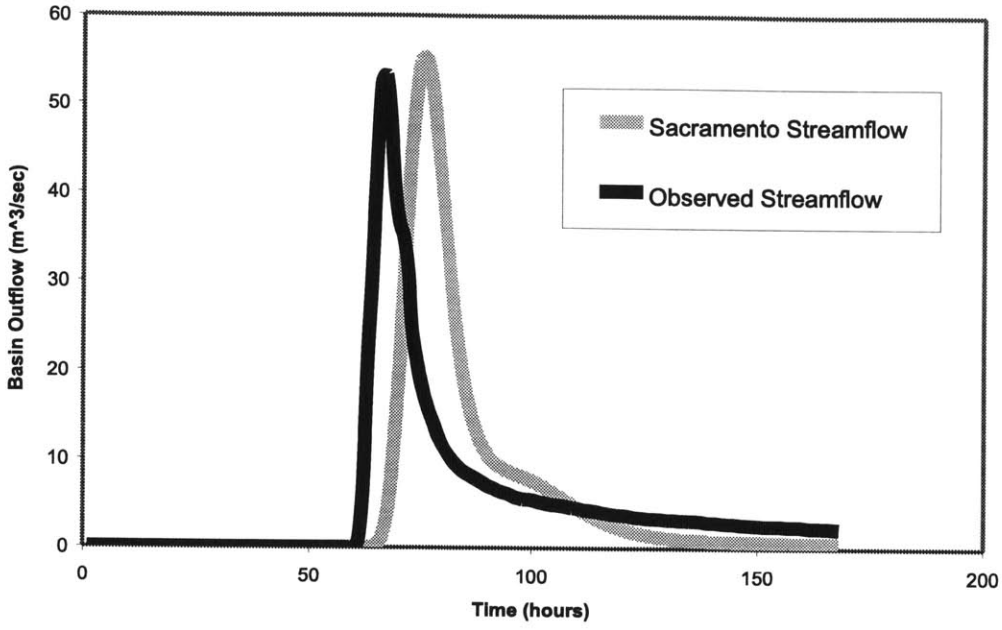


Figure 5-24: Observed and Sacramento Model streamflow for the outlet of the Peacheater Creek Watershed, September 24-30th, 1996

The Sacramento model was also calibrated for the Peacheater Creek watershed. The results of this calibration as developed by the National Weather Service Office of Hydrology were compared to the observed streamflow and these results are given as Figure 5-24.

Chapter 6

Conclusions

As mentioned at the beginning of Chapter 1, this project is only an initial step towards determining the answer to the question of whether distributed models perform better than lumped rainfall/runoff models. The work presented in the previous chapters provides a strong base for future research. The new model framework is a dynamic new system which should allow for both rainfall/runoff modeling and sediment transport. The data collected and manipulated for this project should not be ignored. The dataset provided in Chapter 4 provides an excellent test basin that should be reused in future rainfall/runoff model experiments.

6.1 The tRIBS Model

With the unique integration of the RIBS and CHILD models, an innovative new hydrologic model has been created. Operating under a TIN framework allows for variable resolution in a flood forecasting model. As shown in Chapters 4 and 5, this is significant as the Peacheater Creek watershed is represented effectively with just under 2,500 nodes in tRIBS while the original gridded model requires over 70,000 grid cells to represent the watershed at a 30m cell resolution. This TIN also provides a flexible structure. Streams are located based upon their actual location and are not limited by the size and structure of a grid cell. Another benefit of this new model is its relation to the original CHILD model. A future model incorporating sediment

transport and rainfall/runoff modeling can be derived from tRIBS.

The limited model simulations completed in Chapter 5 demonstrate that tRIBS has the ability to model rainfall/runoff processes. Beginning with a series of hillslope simulations, the model appropriately represents the dynamics of moisture under both storm and interstorm conditions. Mass is conserved in the model and the hydrographs presented are appropriate for each of the three model simulations. The calibration of the Peacheater Creek watershed provides an excellent test of the model capabilities. Using the unique storm event from September 24th–30th, a calibration of the model to actual storm flow was found with a good fit. Under this calibration, all parameters were within the expected ranges for the physical characteristics of the basin. The quality of this calibration event serves as a example of what tRIBS can accomplish as a flood forecasting tool.

6.2 The Peacheater Creek Dataset

As described in Chapter 4, this dataset and the collection of tools used to manipulate the data are a great resource for all rainfall/runoff modelers. As shown, producing a complete dataset for a rainfall/runoff model is a complex task which takes time and a GIS to complete accurately. The use and development of ARC/INFO algorithms was essential to the project allowing for the handling of differing geographic coordinate systems, varying file formats, and data gaps in the inputs used. This same approaches were also used successfully to create the spatially variable data inputs needed in CHILD. With the publication of this data on the project web site, all modelers are encouraged to use this dataset for other rainfall/runoff modeling efforts.

6.3 Future Work

Only a small amount of calibration and no validation have been completed for the tRIBS model. More calibration of tRIBS should be completed in the future to test the robustness of the model. Also, validation of the model using calibration parameters

should be completed as a real test of tRIBS.

The tRIBS model is a fully functional rainfall/runoff model in its current form, but some modifications will improve the model significantly. The parameter describing the rate of conductivity decay, f, should be changed. Modifications to the initialization scheme as presented in Chapter 5 have been made by Ivanov [39] and will be added to the next version of tRIBS. The use of a more complex surface flow algorithm should also be investigated. Many surface flow routing algorithms already exist and they may be appropriate for inclusion in the next version of tRIBS.

At this time, evapotranspiration is given as direct input to the model. A future tRIBS should calculate evapotranspiration from vegatative cover and meteorological inputs. To remove restrictions on the use of tRIBS, algorithms to model snow and reservoir routing could also be added.

As a final note, an attempt to create a comparison between various distributed models along with lumped models has been suggested by the National Weather Service. With the modifications suggested above, tRIBS should be part of this comparison. Its TIN structure is quite unique in the field of rainfall/runoff modeling and should therefore be represented. Also, the Peacheater Creek dataset collected here would be an ideal test basin for this comparison project.

Appendix A

tRIBS User's Guide

This appendix gives all the information needed to compile and run the tRIBS model. Methods for preparing the input to the tRIBS model and viewing the output from the model also also shown here.

A.1 Compiling tRIBS

In the tRIBS directory, there are three sub–directories. The first, Code, has all of the code and make files required for compilation of tRIBS. The make file, rmakeSGI, was developed to compile this program on an SGI machine running IRIX 6.5 or higher and can be executed by typing the following at the prompt within the Code directory:

make -f rmakeSGI

After compiling the model, tRIBS can be run be typing:

tRIBS "inputfile" > "outputfile"

tRIBS can be compiled on other platforms as well. A makefile, rmake, has been provided for compilation on DEC-Alpha systems and since tRIBS uses standard programming structures, compilation on other machines should be fairly simple.

A.2 Creating an input file for tRIBS

An input dataset for tRIBS has a couple of different parts. The first part, is a *.in file which contains the basic information required for the simulation. This text file is the inputfile named when executing the tRIBS model as shown above and it includes information on parameters for the model, simulation time, and links to the distributed datafiles needed for the model. An example of this file is given as peach.in in the tRIBS Code directory.

One other piece of the input dataset is the TIN of the watershed. This is created by following a series of steps. Beginning with the a DEM of the watershed, run the VIP command in ARC/INFO to find a set of interior points for the TIN. Using the boundary of that same watershed, use the BUFFER and UNGENERATE commands in ARC/INFO to create the double ring of points needed for the basin. The points associated with the stream network can then be added using GENERALIZE and UNGENERATE in ARC/INFO. As a final step, createtin.cpp should be run to create the TIN in proper tRIBS format. As example of this is given in the second sub–directory of tRIBS, Input, where the file peach10.input can be found.

To create input rainfall datatsets, tRIBS must first be run for one hour without any distributed input data. This creates the voronoi areas within tRIBS which are needed to convert the data from grids to the specified voronoi areas of the model simulation. The rainfall is then converted from grid to tRIBS format with the use of rainconvert.aml. An example of a final rainfall dataset is given under the Raindata sub–directory of tRIBS.

A.3 Viewing output from tRIBS

The output data from tRIBS has been specially formatted for use in ArcView. Using Tribscov.aml, polygon coverages containing the tRIBS variables are created for easy use within ArcView. The hydrograph of the simulation can be found in the outfile given when executing tRIBS.

Appendix B

tRIBS Model Code

tCNode.h

```
/*************************************************************************\
**
**  tCNode.h
**
**  Header file for derived class tCNode and its member classes.
**   -This is based upon the work of Auroop in the RIBS construct
**
**  Created on February 4th, 1999 by smr
**
**  Last Modified on March 31st, 2000 by smr
\*************************************************************************/

#ifndef TCNODE_H
#define TCNODE_H
#include <iostream.h>
#include <fstream.h>

#include "../MeshElements/meshElements.h"
#include "../tInputFile/tInputFile.h"

//Create a simple class for now

class tCNode : public tNode
{
public:
        tCNode();
        tCNode( tInputFile &infile );
        double getNwtOld();
        double getNwtNew();
        double getMuOld();
        double getMuNew();
        double getMiOld();
        double getMiNew();
        double getNtOld();
        double getNtNew();
        double getNfOld();
        double getNfNew();
        double getRuOld();
        double getRuNew();
        double getRiOld();
        double getRiNew();
        double getQin();
        double getQout();
        double getQpin();
        double getQpout();
        double getRain();
        double getsrf();
        double gethsrf();
        double getesrf();
        double getpsrf();
        double getsatsrf();
        double getrsrf();
        double getsbsrf();
        double getGwaterChng();
        tEdge * getFlowEdg();
        double getTTime();
        int getFloodStatus();
  double getIntStormVar();

        void setMuOld(double);
        void setMuNew(double);
        void setRiOld(double);
        void setRiNew(double);
        void setRuOld(double);
        void setRuNew(double);
        void setNfOld(double);
        void setNfNew(double);
        void setNtOld(double);
```

```
        void setNtNew(double);
        void setNwtOld(double);
        void setNwtNew(double);
        void setMiOld(double);
        void setMiNew(double);
        void setQpout(double);
        void setQpin(double);
        void setRain(double);
        void setsrf(double);
        void sethsrf(double);
        void setesrf(double);
        void setpsrf(double);
        void setsatsrf(double);
        void setrsrf(double);
        void setsbsrf(double);
        void setGwaterChng(double);
        void setFlowEdg(tEdge *);
        void setFloodStatus( int status );
  void setIntStormVar(double);

        void addGwaterChng(double);
  void addQpin(double);
        void addTTime(double);
  void addIntStormVar(double);

        void ActivateSortTracer();
        void MoveSortTracerDownstream();
        void AddTracer();
        int NoMoreTracers();
        tCNode * getDownstrmNbr();
        ~tCNode();
protected:
        double alpha; //slope angle of the node in radians
        double NwtOld, NwtNew; // water table depth in mm
        double MuOld, MuNew; // Moisture Content above WT in mm
        double MiOld, MiNew; // Initialization Moisture above WT in mm
        double NtOld, NtNew; // Top Front in mm
        double NfOld, NfNew; // Wetting Front in mm
        double RuOld, RuNew; // Recharge rate above wetting front in mm
        double RiOld, RiNew; // Recharge rate in mm
        double Qin, Qpin, Qout, Qpout;
        double Rain; // in mm/h;
        double gwchange;
        double srf; // Runoff Generation
        double hsrf; // Hortonian Runoff
        double esrf; // Exfiltration
        double psrf; // Perched Saturation Runoff
        double satsrf; // Groundwater Saturation
        double rsrf; // Return Flow
        double sbsrf; // Saturation from Below runoff
        tEdge * flowedge; //Tells us where this sucker is flowing to.
        int tracer; //used to determine the network structure.
        int flood; //used in sink/fill lake procedures
        double traveltime;
  double intstorm;
};

#endif
```

tCNode.cpp

```
/*****************************************************************************\
**
**  tCNode.cpp
**
**  Functions for derived class tCNode and its member classes
**   -This is based upon the work of Auroop in the RIBS construct
**
**  Created on February 4th, 1999 by smr
**
```

```
**  Last Modified on March 31st, 2000 by smr
\*************************************************************************/

#include <assert.h>
#include <math.h>
#include "../errors/errors.h"
#include "tCNode.h"

tCNode::tCNode()
        :tNode()
{
  alpha = 0.0;
  //NwtOld =  NwtNew = 0.0;
  MuOld = MuNew = 0.0;
        MiOld = MiNew = 0.0;
        NtOld = NtNew = 0.0;
        NfOld = NfNew = 0.0;
        RuOld = RuNew = 0.0;
        RiOld = RiNew = 0.0;
        Qin = Qout = Qpin = Qpout = 0.0;
        Rain = 5.0;
        srf=hsrf=esrf=psrf=satsrf=rsrf=sbsrf=0.0;
        flowedge = 0;
        gwchange = 0.0;
        traveltime = 0.0;
  intstorm = 0.0;
}

tCNode::tCNode(tInputFile &infile)
        : tNode()
{
        alpha = 0.0;
  NwtNew = infile.ReadItem(NwtNew, "WTDEPTH");
        //NwtOld = NwtNew;
  MuOld = MuNew = 0.0;
        MiOld = MiNew = 0.0;
        NtOld = NtNew = 0.0;
        NfOld = NfNew = 0.0;
        RuOld = RuNew = 0.0;
        RiOld = RiNew = 0.0;
        Qin = Qout = Qpin = Qpout = 0.0;
        Rain = infile.ReadItem(Rain, "RAINFALL");
        srf=hsrf=esrf=psrf=satsrf=rsrf=sbsrf=0.0;
        flowedge = 0;
        gwchange = 0.0;
        traveltime = 0.0;
  intstorm = 0.0;
}

//All of the equations to allow for access to the variables!

double tCNode::getNwtOld() {  return NwtOld; }
double tCNode::getNwtNew() {  return NwtNew; }
double tCNode::getMuOld() {  return MuOld; }
double tCNode::getMuNew() {  return MuNew; }
double tCNode::getMiOld() {  return MiOld; }
double tCNode::getMiNew() {  return MiNew; }
double tCNode::getNtOld() {  return NtOld; }
double tCNode::getNtNew() {  return NtNew; }
double tCNode::getNfOld() {  return NfOld; }
double tCNode::getNfNew() {  return NfNew; }
double tCNode::getRuOld() {  return RuOld; }
double tCNode::getRuNew() {  return RuNew; }
double tCNode::getRiOld() {  return RiOld; }
double tCNode::getRiNew() {  return RiNew; }
double tCNode::getQin() {  return Qin; }
double tCNode::getQpin() {  return Qpin; }
double tCNode::getQout() {  return Qout; }
double tCNode::getQpout() {  return Qpout; }
double tCNode::getRain() {  return Rain; }
double tCNode::getsrf() {  return srf; }
```

```
double tCNode::gethsrf() { return hsrf; }
double tCNode::getesrf() { return esrf; }
double tCNode::getpsrf() { return psrf; }
double tCNode::getsatsrf() { return satsrf; }
double tCNode::getrsrf() { return rsrf; }
double tCNode::getsbsrf() { return sbsrf; }
double tCNode::getGwaterChng() { return gwchange; }
tEdge * tCNode::getFlowEdg() { return flowedge; }
double tCNode::getTTime() { return traveltime; }
int tCNode::getFloodStatus() {   return flood; }
double tCNode::getIntStormVar() { return intstorm; }

void tCNode::setMuOld(double value) { MuOld = value; }
void tCNode::setMuNew(double value) { MuNew = value; }
void tCNode::setRiOld(double value) { RiOld = value; }
void tCNode::setRiNew(double value) { RiNew = value; }
void tCNode::setRuOld(double value) { RuOld = value; }
void tCNode::setRuNew(double value) { RuNew = value; }
void tCNode::setNfOld(double value) { NfOld = value; }
void tCNode::setNfNew(double value) { NfNew = value; }
void tCNode::setNtOld(double value) { NtOld = value; }
void tCNode::setNtNew(double value) { NtNew = value; }
void tCNode::setNwtOld(double value) { NwtOld = value; }
void tCNode::setNwtNew(double value) { NwtNew = value; }
void tCNode::setMiOld(double value) { MiOld = value; }
void tCNode::setMiNew(double value) { MiNew = value; }
void tCNode::setQpout(double value) { Qpout = value; }
void tCNode::setQpin(double value) { Qpin = value; }
void tCNode::setRain(double value) { Rain = value; }
void tCNode::setsrf(double value) { srf = value; }
void tCNode::sethsrf(double value) { hsrf = value; }
void tCNode::setesrf(double value) { esrf = value; }
void tCNode::setpsrf(double value) { psrf = value; }
void tCNode::setsatsrf(double value) { satsrf = value; }
void tCNode::setrsrf(double value) { rsrf = value; }
void tCNode::setsbsrf(double value) { sbsrf = value; }
void tCNode::setGwaterChng(double value) { gwchange = value; }
void tCNode::setFlowEdg(tEdge * edg) { flowedge = edg; }
void tCNode::setFloodStatus( int status ) { flood = status; }
void tCNode::setIntStormVar( double value ) { intstorm = value; }

void tCNode::addGwaterChng(double value) {gwchange += value; }
void tCNode::addQpin(double value) { Qpin += value; }
void tCNode::addTTime(double value) { traveltime += value; }
void tCNode::addIntStormVar(double value) { intstorm += value; }

/*************************************************************************\
**
**  Tracer-sorting routines:
**
**  These routines are utilities that are used in sorting the nodes
**  according to their position within the drainage network. The main
**  sorting algorithm is implemented in tStreamNet::SortNodesByNetOrder().
**  The sorting method works by introducing a "tracer" at each point,
**  then allowing the tracers to iteratively cascade downstream. At each
**  step any nodes not containing tracers are moved to the back of the
**  list. The result is a list sorted in upstream-to-downstream order.
**
**  These utilities do the following:
**    ActivateSortTracer -- injects a single tracer at a node
**    AddTracer -- adds a tracer to a node (ignored if node is a bdy)
**    MoveSortTracerDownstream -- removes a tracer and sends it to the
**                                downstream neighbor (unless the node is
**                                a sink; then the tracer just vanishes)
**    NoMoreTracers -- reports whether there are any tracers left here
**
**  Created by GT 12/97. Modified by smr on 3/7/2000
**
\*************************************************************************/

void tCNode::ActivateSortTracer()
```

```
{ tracer = 1; }

void tCNode::MoveSortTracerDownstream()
{
   tracer--;
   getDownstrmNbr()->AddTracer();
}

void tCNode::AddTracer()
{
  //this next statement had to be changed, both 0 & 3 are ok now - SMR

        if((boundary==0)||(boundary==3)) tracer++;
}

int tCNode::NoMoreTracers()
{
   return( tracer==0 );
}

tCNode * tCNode::getDownstrmNbr()
{
   if( flowedge == 0 ) return 0;
   return (tCNode *)flowedge->getDestinationPtrNC();
}

tCNode::~tCNode()
{
  cout << "tCNode has been removed..." << endl;
}
```

tRainfall.h

```
// tRainfall.h
// Created by Scott Rybarczyk on 3/27/2000
// Last modified by smr on 3/27/2000
// Version 0.5
//
// This system handles the rainfall input to the model.
//

#ifndef RAINFALL_H
#define RAINFALL_H

#include "../Definitions.h"
#include "../Classes.h"
#include "../tMesh/tMesh.h"
#include "../tArray/tArray.h"
#include "../tMesh/tMesh.h"
#include "../MeshElements/meshElements.h"
#include "../tInputFile/tInputFile.h"
#include "../tCNode/tCNode.h"
#include "../tRunTimer/tRunTimer.h"

#define kMaxNameSize 80

class tRainfall
{
public:
        tRainfall();
        tRainfall( tMesh<tCNode> *, tInputFile &);
  void NewRain(double curtime);

protected:
  tMesh<tCNode> *gridPtr;
  char inputname[ kMaxNameSize] ;
  ifstream infile;
  tArray<double> rain;
};
```

```
#endif
```

tRainfall.cpp

```
// tRainfall.cpp
// Created by Scott Rybarczyk on 3/27/2000
// Last Modified by smr on 3/27/2000
// Version 0.5
//
// This system handles the rainfall input to the model.
//

#include "tRainfall.h"

tRainfall::tRainfall() {
  gridPtr = 0;
}

tRainfall::tRainfall(tMesh<tCNode> *gridRef, tInputFile &infile) {
  gridPtr = gridRef;
  infile.ReadItem(inputname, "RAINFILE");
  tMeshListIter<tCNode> nodIter(gridPtr->getNodeList());
  cout << "When initializing the rain we have..." <<
    gridPtr->getNodeList()->getSize() << endl;
  tArray<double> rain(gridPtr->getNodeList()->getSize());
  NewRain(0.0);
}

void tRainfall::NewRain(double curtime)
{
  char extension[ 20] ;
  char oneline[ 120] ;
  char *tokenPtr;
  int hour, minute, idno;
  hour = floor(curtime);
  minute = floor((curtime-hour)*60);
  sprintf(extension,"_%d_%d",hour,minute);
  int firsttime;

  rain.setSize(gridPtr->getNodeList()->getSize());

  infile.open(strcat(inputname,extension));
  if(!infile) {
    cerr << "You screwed up, file doesn't exist" << endl;
    exit(1);
  }

  infile >> oneline; // junks the header

  while (infile >> oneline){

    //cout << oneline << endl;

         tokenPtr = strtok( oneline,",");

    firsttime = 0;
         while (tokenPtr != NULL){
      if (firsttime == 0) {
        idno = atoi(tokenPtr);
      }
                    else {
        //cout << "Yep, we are getting here..." << endl;
        rain[ idno]  = atof(tokenPtr);
      }

      //cout << "The rain value is..." << rain[ idno]  << endl;
      //cout << "The idno is..." << idno << endl;
      firsttime++;
      tokenPtr = strtok( NULL, ",");
```

```
    }
  }

  int nodeid;
  tMeshListIter<tCNode> nodeIter( gridPtr->getNodeList() );
  tCNode * cn;

  cn = nodeIter.FirstP();

  while( nodeIter.IsActive())
  {
    nodeid = cn->getID();
    cn->setRain(rain[ nodeid]*10); // converting the rainfall to mm

    cn = nodeIter.NextP();
  }

  cout << "yep we made it here @ " << curtime << endl;
  infile.close();
}
```

tFlowNet.h

```
// tFlowNet.h
// Created by Scott Rybarczyk on 2/13/99
// Last Modified by smr on 2/13/99
// Version 0.5
//
// This is a file for functions that create the flownet of the system.
//      File was created as a small subset of the streamnet.cpp/h system created
//      in the child model...
//
// Note: This object is presently dependent upon tCNodes! Fix this later

#ifndef FLOWNET_H
#define FLOWNET_H

#include "../Definitions.h"
#include "../Classes.h"
#include "../tMesh/tMesh.h"
#include "../tArray/tArray.h"
#include "../tMesh/tMesh.h"
#include "../MeshElements/meshElements.h"
#include "../tInputFile/tInputFile.h"
#include "../tCNode/tCNode.h"
#include "../tRunTimer/tRunTimer.h"
#include "../Array/array1.h"

//As needed for making lakes
#define kFlooded       1  //Part of a lake
#define kNotFlooded    0  //Not part of a lake
#define kCurrentLake   2  //Part of lake being computed
#define kSink          3  //Unfilled depression
#define kOutletFlag    4  //Temp flag in FillLakes
#define kOutletPreFlag 5  // "     "   "     "
#define kVeryHigh 100000  //Used in FillLakes

class tFlowNet
{
public:
        tFlowNet();
        tFlowNet( tMesh< tCNode> *, tInputFile &);
        void CalcSlopes();
        void InitFlowDirs();
        void FlowDirs();
        void SortNodesByNetOrder();
        void setTravelTime();
        void FillLakes();
  int MaxTravel();
  int FindLakeNodeOutlet(tCNode *node);
```

```
        void SurfaceFlow(double curtime, double dt,tArray< double > *);
//      void DumpEnd(double curtime, double dt);
protected:
        tMesh<tCNode> *gridPtr;
  tArray<double> *flowvalue;
  int flowboxes;
        double hillvel;
  double streamvel;
  double velratio;
  double velcoef;
  double baseflow;
  double flowexp;
  double dt;
  double timespan;
        double FlowBox[ 6000];
  double flowout;
  double maxttime;
};

#endif
```

tFlowNet.cpp

```
// tFlowNet.h
// Created by Scott Rybarczyk on 2/13/99
// Last Modified by smr on 12/6/99
// Version 0.5
//
// This is a file for functions that create the flownet of the system.
//      File was created as a small subset of the streamnet.cpp/h system created
//      in the child model...
//
// Note: This object is presently dependent upon tCNodes! Fix this later

#include "tFlowNet.h"
#include "../tArray/tArray.h"

tFlowNet::tFlowNet() {
  gridPtr = 0;
}

tFlowNet::tFlowNet(tMesh<tCNode> *gridRef, tInputFile &infile) {
  gridPtr = gridRef;

  timespan = infile.ReadItem(timespan, "RUNTIME");
  dt = infile.ReadItem(dt, "TIMESTEP");
  velratio = infile.ReadItem(velratio, "VELOCITYRATIO");
  baseflow = infile.ReadItem(baseflow, "BASEFLOW");
  velcoef = infile.ReadItem(velcoef, "VELOCITYCOEF");
  flowexp = infile.ReadItem(flowexp, "FLOWEXP");

  CalcSlopes();
  InitFlowDirs();
  FlowDirs();
  FillLakes();
  SortNodesByNetOrder();
  setTravelTime();
  int i;
//  flowboxes = ceil((maxttime/3600 + timespan)/dt)+5;

//  cout << "In tFlowNet:tFlowNet" << endl;
//  cout << "dt..." << dt << " maxttime..." << maxttime << endl;
//  cout << "timespan..." << timespan << " boxes..." << flowboxes << endl;
//  cout << flush;

//  Array flowvalue(flowboxes);
  flowout = 0.0;
  streamvel = 0.0;
}
```

```
int tFlowNet::MaxTravel()
{
  flowboxes = ceil((maxttime/3600 + timespan)/dt);
  return flowboxes;
}


void tFlowNet::CalcSlopes()
{
  tEdge *curedg;
  tMeshListIter<tEdge> i( gridPtr->getEdgeList() );
  double slp;

  // Loop through each pair of edges on the list
  for( curedg = i.FirstP(); !( i.AtEnd() ); curedg = i.NextP() )
  {
    // Compute the slope and assign it to the current edge
    slp = ( curedg->getOrgZ() - curedg->getDestZ() )
        / curedg->getLength();
    curedg->setSlope( slp );

    // Advance to the edge's complement, and assign it -slp
    curedg = i.NextP();
    curedg->setSlope( -slp );
  }
  //cout << "CalcSlopes() finished" << endl;
}

#define kMaxSpokes 100
void tFlowNet::InitFlowDirs()
{
  tMeshListIter<tCNode> i( gridPtr->getNodeList() );
  tCNode * curnode;
  tEdge * flowedg;
  int ctr;

  // For every active (non-boundary) node, initialize it to flow to a
  // non-boundary node (ie, along a "flowAllowed" edge)
  curnode = i.FirstP();
  while( i.IsActive() )
  {
    // Start with the node's default edge

    flowedg = curnode->getEdg();


    // As long as the current edge is a no-flow edge, advance to the next one
    // counter-clockwise
    ctr = 0;
    while( !flowedg->FlowAllowed() )
    {
      flowedg = flowedg->getCCWEdg();
      ctr++;
      if( ctr>kMaxSpokes ) // Make sure to prevent endless loops
      {
        cerr << "Mesh error: node " << curnode->getID()
             << " appears to be surrounded by closed boundary nodes"
             << endl;
        ReportFatalError( "Bailing out of InitFlowDirs()" );
      }
    }
    curnode->setFlowEdg( flowedg );

    curnode = i.NextP();
  }
}
#undef kMaxSpokes

#define kLargeNegative -1000
#define kMaxSpokes 100
void tFlowNet::FlowDirs()
```

```
{
  tMeshListIter<tCNode> i( gridPtr->getNodeList() );
  double slp;         // steepest slope found so far
  tCNode *curnode;  // ptr to the current node
  //tCNode *newnode;  // ptr to new downstream node
  const tNode *tempnode;  // temporary node for testing purposes
  tEdge * firstedg; // ptr to first edg
  tEdge * curedg;   // pointer to current edge
  tEdge * nbredg;   // steepest neighbouring edge so far
  /*long seed = 91324;
  double chngnum;*/
  int ctr;


  // Find the connected edge with the steepest slope
  curnode = i.FirstP();
  while( i.IsActive() )  // DO for each non-boundary (active) node
  {
    firstedg =  curnode->getFlowEdg();
    slp = firstedg->getSlope();
    nbredg = firstedg;
    curedg = firstedg->getCCWEdg();
    ctr = 0;

    // Check each of the various "spokes", stopping when we've gotten
    // back to the beginning
    while( curedg!=firstedg )
    {
      tempnode=curedg->getOriginPtr();

      if ( curedg->getSlope() > slp && curedg->FlowAllowed() )
      {
        slp = curedg->getSlope();
        nbredg = curedg;
      }
      curedg = curedg->getCCWEdg();
      ctr++;
      if( ctr>kMaxSpokes ) // Make sure to prevent endless loops
      {
        cerr << "Mesh error: node " << curnode->getID()
             << " going round and round" << endl;
        ReportFatalError( "Bailing out of FlowDirs()" );
      }
    }

    curnode->setFlowEdg( nbredg );

    //This is needed to check to pits!-smr

    if( (slp>0) && (curnode->getBoundaryFlag() != kClosedBoundary) )
      curnode->setFloodStatus( kNotFlooded );
    else {
      curnode->setFloodStatus( kSink );
      cout << "This node is a sink->" << curnode->getID() << endl;
    }
    curnode = i.NextP();

  }

  cout << "FlowDirs() finished" << endl << flush;

}
#undef kLargeNegative
#undef kMaxSpokes

void tFlowNet::SortNodesByNetOrder()
{
  int nThisPass;                    // Number moved in current iteration
  int i;
  int done=0;
  tCNode * cn;
```

```
tMeshList<tCNode> *nodeList = gridPtr->getNodeList();
int nUnsortedNodes = nodeList->getActiveSize();  // Number not yet sorted
tMeshListIter<tCNode> listIter( nodeList );

cout << "entering SortNodes by NetOrder()..." << endl;

//test
/*Xcout << "BEFORE: " << endl;
for( cn=listIter.FirstP(); listIter.IsActive(); cn=listIter.NextP() )
  cout << cn->getID() << endl;*/

// Assign initial tracers: use "qs" field, which contains garbage at
// this stage.
for( cn=listIter.FirstP(); listIter.IsActive(); cn=listIter.NextP() )
  cn->ActivateSortTracer();

// Iterate: move tracers downstream and sort until no nodes with tracers
// are left.
do {
// Send tracers downstream
  cn = listIter.FirstP();

  for( i=1; i<=nUnsortedNodes; i++ )
  {
    assert( cn!=0 );
    //cout << "We are at Node #"<<cn->getID()<<" "<<cn->getX()<<" "<<

    //               cn->getY()<<endl;
    cn->MoveSortTracerDownstream();
    //cout << "Downstream Node..." << cn->getDownstrmNbr()->getID() <<endl;
    cn = listIter.NextP();
  }

  // Scan for any nodes that have no tracers, and move them to the bottom
  // of the list.
  tListNode< tCNode > * nodeToMove;
  nThisPass = 0;
  done = TRUE;
  cn = listIter.FirstP();
  for( i=1; i<=nUnsortedNodes; i++ )
  {
    if( cn->NoMoreTracers() )  // If no tracers, move to bottom of list
    {
      nodeToMove = listIter.NodePtr();
      cn = listIter.NextP();
      nodeList->moveToActiveBack( nodeToMove );
      nThisPass++;
    }
    else
    {
      cn = listIter.NextP();
      done = FALSE;
    }
  }

  nUnsortedNodes -= nThisPass;

  //cout << "NO. UNSORTED: " << nUnsortedNodes << endl;
  /*for( cn=listIter.FirstP(); listIter.IsActive(); cn=listIter.NextP() )
      cout << cn->getID() << " " << cn->getQ() << " " << cn->getQs()
           << endl;*/

} while( !done );

/*Xcout << "AFTER: " << endl;
cn = listIter.FirstP();
cout << "First node:\n";
cn->TellAll();
for( cn=listIter.FirstP(); listIter.IsActive(); cn=listIter.NextP() )
    cout << cn->getID() << " " << cn->getQ() << endl;
cout << "Leaving Sort\n" << flush;*/
```

```
}

void tFlowNet::setTravelTime()
{
  cout << "Entering setTravelTime()" << endl;

  tCNode *cn;
  tCNode *ctimer;
  tEdge *ce;
  tMeshListIter<tCNode> nodIter( gridPtr->getNodeList() );
  tMeshListIter<tEdge> edgIter( gridPtr->getEdgeList() );

  // get the velocities first...

  flowout = baseflow + flowout;

  streamvel = velcoef*pow(flowout,flowexp);

  hillvel = streamvel/velratio;

  for( cn=nodIter.FirstP(); nodIter.IsActive(); cn=nodIter.NextP() )
  {
    ctimer = cn;

    while (ctimer->getDownstrmNbr() )
    {
      ce =  ctimer->getFlowEdg();

      // if it is a streamnode, it flows at a stream velocity!

      if (ctimer->getBoundaryFlag() == 3 || ctimer->getBoundaryFlag() == 2)
        cn->addTTime(ce->getLength()/streamvel);
      else
        cn->addTTime(ce->getLength()/hillvel);
      ctimer = ctimer->getDownstrmNbr();
    }
  }

  maxttime = 0.0; // note: the travel times are given in seconds!

  for (cn=nodIter.FirstP(); nodIter.IsActive(); cn=nodIter.NextP() ) {

    if (cn->getTTime() > maxttime) maxttime = cn->getTTime();

    cout << cn->getID() << " " << cn->getX() << " " << cn->getY() << " "
         << cn->getTTime() << " " << cn->getVArea() <<endl;
    }

  cout << "The Maximum Travel Time is ...." << maxttime << endl;
}

void tFlowNet::SurfaceFlow(double curtime, double dt, tArray<double> *flwPtr)
{
  flowvalue = flwPtr;
  tCNode *cn;
  tMeshListIter<tCNode> nodIter( gridPtr->getNodeList() );
  int boxnumber, b;
  double totflow;
  boxnumber = flowvalue->getSize();

  for( cn=nodIter.FirstP(); nodIter.IsActive(); cn=nodIter.NextP() )
  {
    boxnumber = ceil((curtime + cn->getTTime()/3600)/dt); // convert to hours
    //totflow = cn->getsbsrf()*cn->getVArea()/1000; //mm/hr to m^3/h
    totflow = cn->getsrf()*cn->getVArea()/1000; //mm/hr to m^3/hr

// An SMR debug...

if ((boxnumber == 455) && (cn->getsrf() > 0)) {
   cout << "Testable in FlowNet..." << endl;
```

```
    cout << "srf = " << cn->getsrf() << endl;
    cout << "sbsrf = " << cn->getsbsrf() << endl;
    cout << "psrf = " << cn->getpsrf() << endl;
    cout << "hsrf = " << cn->gethsrf() << endl;
    cout << "This occurs @ # " << cn->getID() << endl;
}

     //flowvalue[ boxnumber] = flowvalue[ boxnumber] + totflow;
     flowvalue->addValueTo(boxnumber,totflow);
  }
  //for (b = 0;b < flowboxes; b++)
    //cout << "At time =" << b << "the flow is: " << flowvalue[ b] << endl;
}

//void tFlowNet::DumpEnd(double curtime, double dt)
//{
//      int n;
//      for (n = 0;n<6000; n++)
//    cout << "Flow @ " << curtime+(dt*n) << "is " << FlowBox[ n] << endl;
//}

/*****************************************************************************\
**
**  tStreamNet::FillLakes
**
**  Finds drainage for closed depressions. The algorithm assumes
**  that sinks (nodes that are lower than any of their neighbors)
**  have already been identified during the flow directions
**  procedure. For each sink, the algorithm creates a list of
**  nodes in the current lake, which initially is just the sink
**  itself. It then iteratively looks for the lowest node on the
**  perimeter of the current lake. That node is checked to see
**  whether it can be an outlet, meaning that one of its
**  neighbors is both lower than itself and is not already
**  flooded (or is an open boundary). If the low node is not an
**  outlet, it is added to the current lake and the process
**  is repeated. If it is an outlet, then all of the nodes on the
**  current-lake list are identified draining it. The list is then
**  cleared, and the next sink is processed. If during the search
**  for the lowest node on the perimeter a flooded node is found
**  that isn't already part of the current lake (i.e., it was
**  flagged as a lake node when a previous sink was processed),
**  then it is simply added to the current-lake list --- in other
**  words, the "new" lake absorbs any "old" ones that are encountered.
**
**  Once an outlet has been found, flow directions for nodes in the
**  lake are resolved in order to create a contiguous path through
**  the lake.
**
**    Calls: FindLakeNodeOutlet
**    Called by: MakeFlow
**    Modifies:  flow direction and flood status flag of affected nodes
**    Created: 6/97 GT
**    Modifications:
**     - fixed memory leak on deletion of lakenodes 8/5/97 GT
**     - updated: 12/19/97 SL
**
\*****************************************************************************/
void tFlowNet::FillLakes()
{
   tCNode *cn,              // Node on list: if a sink, then process
       *thenode,            // Node on lake perimeter
       *lowestNode,         // Lowest node on perimeter found so far
       *cln,                // current lake node
       *node;               // placeholder
   tMeshListIter< tCNode > nodIter( gridPtr->getNodeList() ); // node iterator
   tPtrList< tCNode > lakeList;              // List of flooded nodes
   tPtrListIter< tCNode > lakeIter( lakeList ); // Iterator for lake list
   tEdge *ce;               // Pointer to an edge
   double lowestElev;       // Lowest elevation found so far on lake perimeter
   int done;                // Flag indicating whether outlet has been found
```

```
// Check each active node to see whether it is a sink
for( cn = nodIter.FirstP(); nodIter.IsActive(); cn = nodIter.NextP() )
{
   if( cn->getFloodStatus() == kSink )
   {
      // Create a new lake-list, initially containing just the sink node.
      lakeList.insertAtBack( cn );
      cn->setFloodStatus( kCurrentLake );

      // Iteratively search for an outlet along the perimeter of the lake
      done = FALSE;
      do
      {
         lowestNode = lakeIter.FirstP();
         lowestElev = kVeryHigh; // Initialize lowest elev to very high val.

         // Check the neighbors of every node on the lake-list
         for( cln = lakeIter.FirstP(); !( lakeIter.AtEnd() );
              cln = lakeIter.NextP() )
         {
            // Check all the neighbors of the node
            ce = cln->getEdg();
            do
            {
               thenode = (tCNode *) ce->getDestinationPtrNC();
               // Is it a potential outlet (ie, not flooded and not
               // a boundary)?
               if( thenode->getFloodStatus() == kNotFlooded
                   && ce->FlowAllowed() )
               {
                  // Is it lower than the lowest found so far?
                  if( thenode->getZ() < lowestElev )
                  {
                     lowestNode = thenode;
                     lowestElev = thenode->getZ();
                  }
               }
               // If it's a previous lake node or a sink, add it to the list
               else if( thenode->getFloodStatus() == kFlooded ||
                        thenode->getFloodStatus() == kSink )
               {
                  lakeList.insertAtBack( thenode );
                  thenode->setFloodStatus( kCurrentLake );
               }
            } while( ( ce=ce->getCCWEdg() ) != cln->getEdg() );// END spokes

         } /* END lakeList */

         // Now we've found the lowest point on the perimeter. Now test
         // to see whether it's an outlet. If it's an open boundary, it's
         // an outlet...
         if( lowestNode->getBoundaryFlag() == kOpenBoundary ) done = TRUE;
         else // ...it's also an outlet if it can drain to a "dry" location.
         {
            // Can lowestNode drain to a non-flooded location?
            if( FindLakeNodeOutlet( lowestNode ) ) done = TRUE;
            // no, it can't, so add it to the list and continue:
            else
            {
               lakeList.insertAtBack( lowestNode );
               lowestNode->setFloodStatus( kCurrentLake );
            }
         }
         if( lakeList.getSize() > gridPtr->getNodeList()->getActiveSize() )
         {
            cout << "LAKE LIST SIZE: " << lakeList.getSize() << endl;
                                                  if (lakeList.getSize() == 1102)
                                                  {
                                                    for( cln = lakeIter.FirstP(); !(
lakeIter.AtEnd() );
```

```
                    cln = lakeIter.NextP() )
                          {
                                                                    cout << "SMR" << cln->getX()
<< ',' << cln->getY() << endl;
        }
                                                          }
            }

        } while( !done );

        cout << "SMR-Yep we made it here!" << endl;

        // Now we've found an outlet for the current lake.
        // This next bit of code assigns a flowsTo for each node so there's
        // a complete flow path toward the lake's outlet. This isn't strictly
        // necessary --- the nodes could all point directly to the outlet,
        // skipping anything in between --- but it prevents potential problems
        // in ordering the list by network order. This also works by pointing
        // each node toward the first neighboring node they happen to find
        // that has been flagged as having its flow direction resolved.
        // Initially, the low node is thus flagged, and the algorithm repeats
        // until all the lake nodes are flagged as having a flow direction.
        // The algorithm isn't unique---there are many paths that could be
        // taken; this simply finds the most convenient one.
        lowestNode->setFloodStatus( kOutletFlag );

        // Test for error in mesh: if the lowestNode is a closed boundary, it
        // means no outlet can be found.
        do
        {
           done = TRUE;  // assume done until proven otherwise
           for( cln = lakeIter.FirstP(); !( lakeIter.AtEnd() );
                cln = lakeIter.NextP() )
           {
              if( cln->getFloodStatus() != kOutletFlag )
              {
                 done = FALSE;

                 // Check each neighbor
                 ce = cln->getEdg();
                 do
                 {
                    node = (tCNode *) ce->getDestinationPtrNC();
                    if( node->getFloodStatus() == kOutletFlag )
                    {      // found one!
                       cln->setFloodStatus( kOutletPreFlag );
                       cln->setFlowEdg( ce );
                       //cout << "Node " << cln->getID() << " flows to "
                       //<< cln->getDownstrmNbr()->getID() << endl;

                    }
                 } while( cln->getFloodStatus() != kOutletFlag
                          && ( ce=ce->getCCWEdg() ) != cln->getEdg() );
              } // END if node not flagged as outlet
           } // END for each lake node

           // Now flag all the "preflagged" lake nodes as outlets
           for( cln = lakeIter.FirstP(); !( lakeIter.AtEnd() );
                cln = lakeIter.NextP() )
               if( cln->getFloodStatus() == kOutletPreFlag )
                   cln->setFloodStatus( kOutletFlag );

        } while( !done );
        lowestNode->setFloodStatus( kNotFlooded );

        // Finally, flag all of the
        // nodes in it as "kFlooded" and clear the list so we can move on to
        // the next sink. (Fixed mem leak here 8/5/97 GT).
        for( cln = lakeIter.FirstP(); !( lakeIter.AtEnd() );
             cln = lakeIter.NextP() )
            cln->setFloodStatus( kFlooded );
```

```
         lakeList.Flush();
      } /* END if Sink */
   } /* END Active Nodes */
   //cout << "FillLakes() finished" << endl << flush;

} // end of tFlowNet::FillLakes


/*****************************************************************************\
**
**  FindLakeNodeOutlet
**
**  This function is part of the lake-filling algorithm. It checks to see
**  whether there is a valid outlet for the current node, and if so it
**  assigns that outlet. An "outlet" essentially means a downhill neighbor
**  that isn't already flooded to the level of the current node. The function
**  performs basically the same operation as FlowDirs, but with stricter
**  criteria. The criteria for a valid outlet are:
**
**  (1) It must be lower than the current node (slope > 0)
**  (2) It must not be part of the current lake (a lake can't outlet to itself)
**  (3) It must not be a closed boundary (_flowAllowed_ must be TRUE)
**  (4) If the outlet is itself part of a different lake, the water surface
**      elevation of that lake must be lower than the current node.
**
**  Returns: TRUE if a valid outlet is found, FALSE otherwise
**  Calls: (none)
**  Called by: FillLakes
**  Created: 6/97 GT
**  Updated: 12/19/97 SL; 1/15/98 gt bug fix (open boundary condition)
**
\*****************************************************************************/
int tFlowNet::FindLakeNodeOutlet( tCNode *node )
{
   double maxslp = 0;  // Maximum slope found so far
   tEdge * ce;         // Current edge
   //XtPtrListIter< tEdge > spokIter( node->getSpokeListNC() );
   tCNode *dn,         // Potential outlet
       *an;            // Node ptr used to find outlet of a previously
                       // identified lake

   // Check all the neighbors
   ce = node->getEdg();
   do
   {
      // If it passes this test, it's a valid outlet
      dn = (tCNode *) ce->getDestinationPtrNC();
      assert( dn>0 );
/*X       if( ce->getSlope() > maxslp &&
          dn->getFloodStatus() != kCurrentLake &&
          ce->FlowAllowed() &&
          ( dn->getBoundaryFlag()==kOpenBoundary ||
            dn->getDownstrmNbr()->getZ() < node->getZ() ) )*/
      if( ce->getSlope() > maxslp &&
          dn->getFloodStatus() != kCurrentLake &&
          ce->FlowAllowed() )
      {
         // Handle a very special and rare case: if the "target" node dn is
         // part of a previous lake, it's still a valid exit as long as its
         // water surface elevation is lower than the current lake (whose
         // wse, assuming an outlet is found, would be equal to _node_'s
         // elevation). It can sometimes happen that the target lake's wse is
         // exactly equal in elevation to _node_, in which case
         // the point is not considered an outlet---if it were, infinite loops
         // could result. (This fix added 4/98)
         if( dn->getFloodStatus()==kFlooded )
         {
            // Iterate "downstream" through the "old" lake until reaching the
            // outlet, then test its elevation. If the elevation is exactly
            // equal to _node_, skip the rest and go on to the next iteration.
            an = dn;
```

```
            while( an->getFloodStatus()!=kNotFlooded )
                an = an->getDownstrmNbr();
            if( an->getZ()==node->getZ() ) continue;
          }

          // Assign the new max slope and set the flow edge accordingly
          maxslp = ce->getSlope();
          node->setFlowEdg( ce );
          // cout << "Node " << node->getID() << " flows to "
          //      << node->getDownstrmNbr()->getID() << endl;
        }
    } while( ( ce=ce->getCCWEdg() ) != node->getEdg() );

    return( maxslp > 0 );
}
```

Cbsim.h

```
// cbsim.h
//
// Created by Scott Rybarczyk on 2/4/99
// Last Modified by smr on 5/17/00
// Version 1.0
//
// This is a header file for objects related to groundwater transport
//
// Note: This object is presently dependent upon tCNodes!

#ifndef CBSIM_H
#define CBSIM_H

#include "../Definitions.h"
#include "../Classes.h"
#include "../tMesh/tMesh.h"
#include "../tArray/tArray.h"
#include "../tInputFile/tInputFile.h"
#include "../tCNode/tCNode.h"
#include "../tRunTimer/tRunTimer.h"

#define kMaxNameSize 80

class Cbsim
{
public:
        Cbsim( tMesh < tCNode > *, tInputFile &);
  void Auroop(double dt);
        void Reset();
        void InitSet();
        void GndWater(double dt);
        double LambertW(double z);
  char gwatfile[ kMaxNameSize] ;

protected:
  tArray<double> gwaterval;


private:
        double ksat;
        double f;
        double theta_s;
        double theta_r;
        double epsilon;
        double Ar;
        double UAr;
        double porosity;
  double IntStormMax;
  double PoreInd;
  double Psib;

        tMesh<tCNode> *gridPtr;
```

```
};

#endif
```

Cbsim.cpp

```
// cbsim.cpp
// Created by Scott Rybarczyk on 2/4/99
// Last Modified by smr on 6/20/00
// Version 0.5
//
// This is a file for objects related to groundwater transport
//
// Note: This object is presently dependent upon tCNodes! Fix this later
// ***This object is the one modified by Valeri's Code...
//
// The constructor for cbsim! Must be given the input file, and the list of
// nodes. Right now, they must be tCNodes...
//

#include <math.h>
#include <assert.h>
#include <iomanip.h>
#include "cbsim.h"
#include "complex1.c"

#define EPS 2.2204e-16

Cbsim::Cbsim(tMesh<tCNode> *gptr, tInputFile &infile)
{
  gridPtr = gptr;
  ksat = infile.ReadItem(ksat, "KSAT");
  f = infile.ReadItem(f, "EFOLD");
  theta_s = infile.ReadItem(theta_s, "THETASAT");
  theta_r = infile.ReadItem(theta_r, "THETARES");
//  epsilon = infile.ReadItem(epsilon, "EPSILON" );
  Ar = infile.ReadItem(Ar, "ANIRATIO");
  UAr = infile.ReadItem(UAr, "UNSATAR");
  porosity = infile.ReadItem(porosity, "POROSITY" );
  IntStormMax = infile.ReadItem(IntStormMax, "INTSTORMMAX" );
  Psib = infile.ReadItem(Psib, "BUBBPRES" );
  PoreInd = infile.ReadItem(PoreInd, "POREINDEX" );
  infile.ReadItem(gwatfile, "GWATERFILE" );
  epsilon = 3 + 2/PoreInd;
}

void Cbsim::Auroop(double dt)
{
  tCNode * cn;
  tCNode * dnode;
  tCNode * cdest;
  tEdge * ce;
  tMeshListIter<tCNode> nodIter( gridPtr->getNodeList() );
  tMeshListIter<tCNode> unsortIter( gridPtr->getUnsortList() );
  tMeshListIter<tEdge> edgIter( gridPtr->getEdgeList() );
  double thetasur;
  double alpha, Cos, Sin;
  double Kunsat;
  double Ractual;
  double recharge;
  double Mdelt, Mdva, AA, BB; //viva variables
  int Pixel_State;
  double xxsrf; //used for ease of coding
  double Mperch; //again, an ease of coding thing.
  double Nstar, ThRiNf, ThRiNstar, ThReNf, ThRuNt, qn;
  double temphold;
  double NwtNext, NfNext;
  double RADAR = 0.1; //checks to see if we are in evaporation (in mm/h)
  double SeIn, Se0, G; // ponded variables...
```

```
cout << "This testing loop can be found within cbsim.cpp-smr" << endl <<
          flush;

enum { Storm_Evol, WTStaysAtSurf,WTDropsFromSurf,WTGetsToSurf,Storm_Unsat_Evol,
  Perched_Evol, Perched_SurfSat, StormToInterTransition, ExactInitial,
  IntStormBelow} ;

cn = nodIter.FirstP();

while( nodIter.IsActive() )
{
  cn->setQpin(0.0);
  cn = nodIter.NextP();
}

cn = nodIter.FirstP();

while( nodIter.IsActive() )
{

  //step1: compute k_unsaturated and rainfall stuff

  if (ksat!=0)
    thetasur=pow((cn->getRiOld()/ksat),(1/epsilon))*(theta_s-theta_r)+theta_r;

  if (cn->getRuOld()==0)
    Kunsat = cn->getRiOld();
  else
    Kunsat = cn->getRuOld();

  ce = cn->getFlowEdg();
  alpha = atan(ce->getSlope());
  Cos = cos(alpha);
  Sin = sin(alpha);

  Ractual = cn->getRain();
  recharge = (Ractual + cn->getQpin() - cn->getQpout())*Cos;

  //step2: determine the node state!

  // ************************************************
  Pixel_State=-1000;
  if ( (cn->getNwtOld()==0) && (recharge>=0) )  Pixel_State = WTStaysAtSurf;
        // WT initially at surface & stays there

  if ( (cn->getNwtOld()==0) && (recharge<0) )  Pixel_State = WTDropsFromSurf;
       // WT initially at surface & drops

  if ((cn->getNwtOld()>0) && ((recharge*dt) >= (cn->getNwtOld()*theta_s -
    cn->getMuOld()))&&(cn->getNfOld()==0 || cn->getNfOld()==cn->getNwtOld()))
      Pixel_State = WTGetsToSurf;
       // WT initially at some depth reaches surface

  // ************************************************

  if (Pixel_State==-1000) {      // None of the previous cases is applied..
    cn->setMuNew(cn->getMuOld() + dt*recharge);

    if (cn->getMuNew() < cn->getMiOld())
           Pixel_State = IntStormBelow;   // apply interstorm eqn.

    if (cn->getMuNew() == cn->getMiOld())
           Pixel_State = ExactInitial;    // Exactly Initialized State

    if (cn->getMuNew() > cn->getMiOld())  {
      if (recharge > 0) {
        if (cn->getNtOld() == cn->getNfOld())
          Pixel_State = Storm_Unsat_Evol;

        if ((cn->getNtOld() < cn->getNfOld()) && (cn->getNtOld() != 0))
          Pixel_State = Perched_Evol;
```

```
        if ((cn->getNtOld()==0) && (cn->getNfOld() > 0)) {
          ThRiNf = theta_r + exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)
             *pow((cn->getRiOld()/ksat),(1/epsilon));
          SeIn = pow(((ThRiNf-theta_r)/(theta_s-theta_r)),(3+1/PoreInd));
          G = -Psib/PoreInd*(1-SeIn)/(3+1/PoreInd)*(1-exp(-f*cn->getNfOld()))/
                 (f*cn->getNfOld());

          qn = ksat*f*cn->getNfOld()/(exp(f*cn->getNfOld())-1)
                  *(Cos+G/cn->getNfOld());
          xxsrf = qn - cn->getRiOld()*Cos; // net infiltration
          if (recharge >= xxsrf) Pixel_State = Perched_SurfSat;
          else Pixel_State = StormToInterTransition;
        }

        // The following is still fairly new and needs
        // to be checked. It should handle the following...
        // a) Nf has reached wt with continued rainfall->redistribute water
        // b) Subsurface lateral flows should not form new Nf!

        if ((cn->getNfOld() == cn->getNwtOld()) || (cn->getRain() <= RADAR &&
          cn->getIntStormVar() > 0 && cn->getNfOld() == 0))
          Pixel_State = Storm_Evol;  // Totally New Pixel State
        }
        if (recharge <= 0) Pixel_State = StormToInterTransition;
      }
      if (ksat==0) Pixel_State = Perched_SurfSat;
    }


// End of "case" definitions
//step 3: define the cases!

switch (Pixel_State) {

  case WTStaysAtSurf:
    if (cn->getID()==600) //SMR debug
      cout << "We are in WTStaysAtSurf!" << '\n';
    //seperation into psrf & sbsrf depending upon recharge vs ksat

    cn->setsbsrf(cn->getRain()*Cos);

    //Exfiltration occurs due to lateral inflows
    if (cn->getQpin() > cn->getQpout())
      cn->setpsrf((cn->getQpin() - cn->getQpout())*Cos);
    else
      cn->setsbsrf(cn->getsbsrf() - (cn->getQpin() - cn->getQpout())*Cos);

    cn->setMuNew(0.0);
    cn->setRiNew(0.0);
    cn->setRuNew(0.0);
    cn->setNfNew(0.0);
    cn->setNtNew(0.0);
    cn->setNwtNew(0.0);
    cn->setMiNew(0.0);
    cn->setQpout(0.0);

    if (cn->gethsrf() < 0) cout << "WARNING! hsrf runoff < 0!" << endl;
    if (cn->getsbsrf() < 0) cout << "WARNING! sbsrf runoff < 0!" << endl;
    if (cn->getpsrf() < 0) cout << "WARNING! psrf runoff < 0!" << endl;

    if (cn->getRain()<= RADAR) cn->addIntStormVar(dt);
    else {
      if (cn->getIntStormVar() > 0) cn->setIntStormVar(0.0);
    }

  break;

  case WTGetsToSurf:
    if (cn->getID()==600) // SMR debug
      cout << "We are in WTGetsToSurf!" << '\n';
```

```
  if (cn->getQpin() > cn->getQpout()) {
    cn->setpsrf((cn->getQpin()-cn->getQpout())*Cos+cn->getMuOld()/dt -
                cn->getNwtOld()*theta_s/dt);
    if (cn->getpsrf() < 0) {
      cn->setsbsrf(cn->getRain()*Cos - cn->gethsrf() + cn->getpsrf());
      cn->setpsrf(0.0);
    }
    else
      cn->setsbsrf(cn->getRain()*Cos - cn->gethsrf());
  }
  else
    cn->setsbsrf((cn->getRain()+cn->getQpin()-cn->getQpout())*Cos-
        cn->gethsrf()+cn->getMuOld()/dt - cn->getNwtOld()*theta_s/dt);

  cn->setMuNew(0.0);
  cn->setRiNew(0.0);
  cn->setRuNew(0.0);
  cn->setNfNew(0.0);
  cn->setNtNew(0.0);
  cn->setNwtNew(0.0);
  cn->setMiNew(0.0);
  cn->setQpout(0.0);

  if (cn->gethsrf() < 0) cout << "WARNING! hsrf runoff < 0!" << endl;
  if (cn->getsbsrf() < 0) cout << "WARNING! sbsrf runoff < 0!" << endl;
  if (cn->getpsrf() < 0) cout << "WARNING! psrf runoff < 0!" << endl;

  if (cn->getIntStormVar() > 0) cn->setIntStormVar(0.0);

break;

case WTDropsFromSurf:

// This algorithm has been totally revamped by Valeri.

  if (cn->getID()==600) // SMR debug
    cout << "We are in WTDropsFromSurf!" << '\n';

  recharge = (cn->getRain()+cn->getQpin()-cn->getQpout())*Cos;
  if (recharge < 0) {
    BB = -exp(f*recharge/(epsilon*(theta_s-theta_r))-1);
    cn->setNwtNew(epsilon/f*(1+ LambertW(BB))-recharge/(theta_s-theta_r));
    if (BB < (-1/exp(1)))
      cout << "Value for Lambert Function too small! - WTDFS" << endl;
  }

  temphold = epsilon/f*(theta_s-theta_r)*
    (1-exp(-f*cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew();
  cn->setMiNew(temphold);
  cn->setMuNew(cn->getMiNew());
  cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
  cn->setRuNew(0.0);
  cn->setQpout(0.0);

  cn->addIntStormVar(dt);

  if (cn->getIntStormVar() >= IntStormMax) {
    cn->setNfNew(0.0);
    cn->setNtNew(0.0);
  }
  else {
    cn->setNfNew(cn->getNwtNew());
    cn->setNtNew(cn->getNwtNew());
  }

break;

case ExactInitial:
  if (cn->getID()==600) // SMR Debug!
    cout << "We are in ExactInitial!" << '\n';
```

```
  //This shouldn't really happen, would need to be a very weird event

  cn->setMiNew(cn->getMiOld());
  cn->setMuNew(cn->getMiNew());
  cn->setNwtNew(cn->getNwtOld());
  cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
  cn->setRuNew(0.0);
  cn->setQpout(0.0);

  if (cn->getRain() <= 0) cn->addIntStormVar(dt);

  if (cn->getNfOld()==cn->getNwtOld()) {
    if (cn->getIntStormVar() >= IntStormMax) {
      cn->setNfNew(0.0);
      cn->setNtNew(0.0);
    }
    else {
      cn->setNfNew(cn->getNwtNew());
      cn->setNtNew(cn->getNwtNew());
    }
  }
  else {
    cn->setNfNew(0.0);
    cn->setNtNew(0.0);
  }

break;

case IntStormBelow:
  if (cn->getID()==600) // SMR Debug!
    cout << "We are in IntStormBelow!" << '\n';

  recharge = (cn->getRain()+cn->getQpin()-cn->getQpout())*Cos;
  cn->setMuNew(cn->getMuOld() + recharge*dt);
  AA = cn->getMuNew()-epsilon/f*(theta_s-theta_r)-theta_s*cn->getNwtOld();
  BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
  if (BB< (-1/exp(1)))
    cout << "Value for Lambert Function too small! - ISB" << endl;
  cn->setNwtNew(epsilon/f*LambertW(BB)-AA/(theta_s-theta_r));
  cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
  cn->setRuNew(0.0);
  cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNwtNew()/
               epsilon)) + theta_r*cn->getNwtNew());
  cn->setMuNew(cn->getMiNew());
  cn->setQpout(0.0);

  cn->addIntStormVar(dt);

  if ((cn->getNfOld() < cn->getNwtOld()) && (cn->getNfOld() > 0)){
    cn->setNfNew(0.0);
    cn->setNtNew(0.0);
  }
  else {
    if (cn->getIntStormVar() < IntStormMax) {
      if (cn->getNfOld()==0) {
        cn->setNfNew(0.0);
        cn->setNtNew(0.0);
      }
      else {
        cn->setNfNew(cn->getNwtNew());
        cn->setNtNew(cn->getNwtNew());
      }

    }
    else if (cn->getIntStormVar() >= IntStormMax) {
      cn->setNfNew(0.0);
      cn->setNtNew(0.0);
    }
  }

break;
```

```
    case StormToInterTransition:
      if (cn->getID()==600) // SMR Debug!
        cout << "We are in StormToInterTransition!" << '\n';

      if (cn->getRain()<=RADAR)
        cn->addIntStormVar(dt);
      else
        cn->setIntStormVar(0.0);

      recharge = (cn->getRain()+ cn->getQpin() - cn->getQpout())*Cos;
      cn->setMuNew(cn->getMuOld() + recharge*dt);
      cn->setNwtNew(cn->getNwtOld());
      cn->setMiNew(cn->getMiOld());
      cn->setRiNew(cn->getRiOld());

      if (cn->getIntStormVar()>=IntStormMax) { //destroy edge & redist. water
        AA = cn->getMuNew() - epsilon/f*(theta_s-theta_r) -
                theta_s*cn->getNwtOld();
        BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
        if (BB< (-1/exp(1))) {
          cout << "Value for Lambert Function too small! - SIT" << endl;
          cout << "Id = " << cn->getID() << endl;
          cout << "MuNew = " << cn->getMuNew() << endl;
          cout << "NwtOld = " << cn->getNwtOld() << endl;
          cout << "AA = " << AA << endl;
          cout << "BB = " << BB << endl;
        }

        cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
        cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
        cn->setRuNew(0.0);
        cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNwtNew()/
              epsilon)) + theta_r*cn->getNwtNew());
        cn->setMuNew(cn->getMiNew());
        cn->setNfNew(0.0);
        cn->setNtNew(0.0);
        cn->setQpout(0.0);
      }

      else { // the edge is still moving down...

      Mdelt = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*(1-exp(f*
              (cn->getNfOld()-cn->getNwtNew())
              /epsilon)) + theta_r*(cn->getNwtNew()-cn->getNfOld()));
      // <--- amount of water in the wetted edge..
      AA = epsilon/f*(theta_s-theta_r)*(1 - exp(-f*cn->getNfOld()/epsilon)) +
theta_r*cn->getNfOld();
      // <--- maximum moisture in the edge allowed to be in UNSATURATED STATE

      // We now use an artificial algorithm to correct the variables.


      if (Mdelt > AA && cn->getNtOld() == cn->getNfOld()) {
        cout << "Imbalance Case: Corrected by " << Mdelt - AA + 1e-4 << endl;
        Mdelt = AA - 1e-4;
      }

      // ----------------------
      // |###### TYPE 1 ######|
      // ----------------------

      if (Mdelt < AA) {   //  <========== UNSATURATED EDGE!!!

        if (recharge == 0) {
          cn->setRuNew(cn->getRuOld());
          Nstar = (log(ksat/cn->getRuNew()))/f;
        }
        else {
          Mdelt = (Mdelt - theta_r*cn->getNfOld())/(theta_s-theta_r);
          Mdelt = Mdelt*f/(epsilon*(exp(f*cn->getNfOld()/epsilon) - 1));
```

```
  cn->setRuNew(ksat*pow(Mdelt,epsilon));
  Nstar = (log(ksat/cn->getRuNew()))/f;
  if (Nstar>cn->getNwtNew())
    cout << "WARNING!!! Nstar > WT depth < 0" << endl;
}

// ----> Move fronts if necessary

if ((100*(cn->getMuNew()-cn->getMiOld())/cn->getMiOld()) <= 0.10) {
  // Edge gets very close to the MiOld profile, erase it!
  cn->setNfNew(0.0);
  cn->setNtNew(0.0);

  if (cn->getMuNew()-cn->getMiOld() > 1e-4) {
    AA = cn->getMuNew() - epsilon/f*(theta_s-theta_r)
           - theta_s*cn->getNwtOld();
    BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
    if (BB < (-1/exp(1)))
      cout << "Value for Lambert function is too small - SIT2" << endl;
    cn->setNwtNew(epsilon/f*LambertW(BB)-AA/(theta_s-theta_r));
    cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
    cn->setRuNew(0.0);
    cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1 - exp(-f*cn->getNwtNew()
          /epsilon)) + theta_r*cn->getNwtNew());
    cn->setMuNew(cn->getMiNew());
  }
  else {    // <===== INSIGNIFICANT amount of moisture
    cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1 - exp(-f*
          cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew());
    cn->setMuNew(cn->getMiNew());
    cn->setRuNew(0.0);
  }

}
else {    // <--- edge is still significant (RuNew>RiNew >> 0.10%)
  ThRiNf = theta_r + exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)
            *pow((cn->getRiNew()/ksat),(1/epsilon));
  ThReNf = theta_r + exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)
            *pow((cn->getRuNew()/ksat),(1/epsilon));
  SeIn = pow( ((ThRiNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
  Se0  = pow( ((ThReNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
       G = -Psib/PoreInd*(Se0 - SeIn)/(3 + 1/PoreInd);
  G = G*(1-exp(-f*cn->getNfOld()))/(f*cn->getNfOld());
  qn = cn->getRuNew()*Cos+ksat*exp(-f*cn->getNfOld())*G/cn->getNfOld();

  cn->setNfNew(cn->getNfOld()+dt*(qn-cn->getRiNew())
            *Cos/(ThReNf-ThRiNf));
  cn->setNtNew(cn->getNfNew());

  Mdelt = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*(1-exp
          (f*(cn->getNfNew() - cn->getNwtNew())/epsilon)) +
          theta_r*(cn->getNwtNew()-cn->getNfNew()));
  AA = epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNfNew()/epsilon))+
           theta_r*cn->getNfNew();
  BB = (Mdelt - theta_r*cn->getNfNew())/(theta_s-theta_r);
  BB = BB*f/(epsilon*(exp(f*cn->getNfNew()/epsilon) - 1));
  cn->setRuNew(ksat*pow(BB,epsilon));
  // <=== recharge rate after redistribution

  if (Mdelt > AA) {
    if ((Mdelt-AA) > 10.0)
      cout << "Incorrect dynamics in StormToInterTransition!" << endl;
    cn->setNtNew((log(ksat/cn->getRuNew()))/f);
  }

  // CHECK, if RECHARGE RATE is very small:
  // if so, just redistribute this saturated edge!

  if (cn->getNfNew() > cn->getNwtNew()) {
    if ((cn->getMuNew() - cn->getMiOld()) > 1.0e-4) {
      AA = cn->getMuNew() - epsilon/f*(theta_s-theta_r) -
```

```
            theta_s*cn->getNwtOld();
        BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
        if (BB < (-1/exp(1)))
          cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
        cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
        cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
        cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1 -
          exp(-f*cn->getNwtNew()/epsilon))+theta_r*cn->getNwtNew());
        cn->setMuNew(cn->getMiNew());
      }
      else {    // <===== INSIGNIFICANT amount of moisture
        cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*
            cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew());
        cn->setMuNew(cn->getMiNew());
      }

      cdest = (tCNode*)ce->getDestinationPtrNC();
      NwtNext = cdest->getNwtOld();
      NfNext = cdest->getNfOld();

      if ((NfNext == NwtNext) && (cn->getIntStormVar() < IntStormMax)) {
        cn->setNfNew(cn->getNwtNew());
        cn->setNtNew(cn->getNwtNew());
      }
      else {
        cn->setNfNew(0.0);
        cn->setNtNew(0.0);
      }
      cn->setRuNew(0.0);
    }
  }
}  // <======== End of TYPE 1

// ELSE here includes evolution of perch & surface-saturated zone...

else {

// ----------------------
// |###### TYPE 2 ######|
// ----------------------

// This includes evolution of perch & surface-saturated zone...

  //    --------------------------------
  //           move wetting front
  //    --------------------------------

  ThRiNf = theta_r+exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)
           *pow((cn->getRiNew()/ksat),(1/epsilon));

  SeIn = pow( ((ThRiNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
  G = -Psib/PoreInd*(1 - SeIn)/(3 + 1/PoreInd);
  G = G*(1-exp(-f*cn->getNfOld()))/(f*cn->getNfOld());
  qn = ksat*f*(cn->getNfOld()-cn->getNtOld())/(exp(f*cn->getNfOld())
         -exp(f*cn->getNtOld()));
  qn *= (Cos + G/cn->getNfOld());

  cn->setNfNew(cn->getNfOld()+dt*(qn-cn->getRiNew()*Cos)
               /(theta_s-ThRiNf));


  //  ************************************************************
  //     C H E C K  for possible situations with Nf & Nwt....
  //  ************************************************************

  if (fabs(cn->getNfNew() - cn->getNwtNew()) <= 1.0e-3) {
    cn->setNwtNew(cn->getNtOld());
    // <=== this will be my flag for Ntop dynamics..

    cdest = (tCNode*)ce->getDestinationPtrNC();
    NwtNext = cdest->getNwtOld();
```

```
  NfNext = cdest->getNfOld();

  if ((NfNext == NwtNext) && (cn->getIntStormVar() < IntStormMax))
    cn->setNfNew(cn->getNwtNew());
  else
    cn->setNfNew(0.0);
}
else if (cn->getNfNew() > cn->getNwtNew()) {
  cn->setNwtNew(cn->getNtOld());  // <=== this flags Ntop dynamics..

  cdest = (tCNode*)ce->getDestinationPtrNC();
  NwtNext = cdest->getNwtOld();
  NfNext = cdest->getNfOld();

  if ((NfNext == NwtNext) && (cn->getIntStormVar() < IntStormMax))
    cn->setNfNew(cn->getNwtNew());
  else
    cn->setNfNew(0.0);

  // *** increase recharge by excess flow... ***
  recharge += qn - ((cn->getNwtOld()-cn->getNfOld())*(theta_s-ThRiNf)/dt
    + cn->getRiNew()*Cos);
}

//      --------------------------------
//               move top front
//      --------------------------------

if ((cn->getNfNew() == 0) || (cn->getNfNew() == cn->getNwtNew())) {
//  <--- Wetting front hits the water table
  if (cn->getMuNew() >= cn->getNwtOld()*theta_s) {
    cn->setNwtNew(0.0);
    cn->setRiNew(0.0);
    cn->setRuNew(0.0);
    cn->setMiNew(0.0);
    cn->setNfNew(0.0);
    cn->setNtNew(0.0);
    cn->setsbsrf((cn->getMuNew()-cn->getNwtOld()*theta_s)/dt);
    cn->setMuNew(0.0);
  }
  else {
    BB = -dt*(qn - (recharge+cn->getRiNew()*Cos))/(theta_s-theta_r) -
         epsilon/f * exp(-f*cn->getNtOld()/epsilon);
    AA = -exp(f*(BB-cn->getNtOld())/epsilon);
    if (AA < (-1/exp(1)))
      cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
    cn->setNtNew(cn->getNtOld()+(epsilon/f*LambertW(AA) - BB));

    if (cn->getNfNew() == cn->getNwtNew()) {
      cn->setNwtNew(cn->getNtNew());
      cn->setNfNew(cn->getNtNew());
    }
    else {
      cn->setNwtNew(cn->getNtNew());
      cn->setNtNew(0.0);
      cn->setNfNew(0.0);
    }

    cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
    cn->setRuNew(0.0);
    cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNwtNew()
                /epsilon)) + theta_r*cn->getNwtNew());
    cn->setMuNew(cn->getMiNew());
  }
}
else if ((cn->getNfNew() > 0) && (cn->getNfNew() != cn->getNwtNew())){
//  ...still above the water table...
  Mdelt = epsilon/f*(theta_s-theta_r)*(1 -
          exp(-f*cn->getNtOld()/epsilon)) + theta_r*cn->getNtOld();
  Mdva  = epsilon/f*(theta_s-theta_r)*(1 -
          exp(-f*cn->getNfNew()/epsilon)) + theta_r*cn->getNfNew();
```

```
            AA = (Mdelt+(cn->getNfNew()-cn->getNtOld())*theta_s - Mdva);
          // Max amt. of water that can be lost w/o unsaturating pixel
          if ((recharge+cn->getRiNew()*Cos) < (qn - AA/dt)) {
            // <--- Next state is UnSaturated!!!
            Mdelt = cn->getMuNew() - (epsilon/f*(theta_s-theta_r)*(1-exp(f*(
               cn->getNfNew()-cn->getNwtNew())/epsilon))+ theta_r*
               (cn->getNwtNew()-cn->getNfNew()));
            Mperch = (Mdelt-theta_r*cn->getNfNew())/(theta_s-theta_r);
            Mperch = Mperch*f/(epsilon*(exp(f*cn->getNfNew()/epsilon)-1));
            cn->setRuNew(ksat*pow(Mperch,epsilon));
            //  <== Equiv. recharge rate above Nf (approxim.)
            cn->setNtNew(cn->getNfNew());

            if (Mdelt >= Mdva) { // needed for numerical accuracy
              cn->setMuNew(cn->getMuNew()-(Mdelt-Mdva + 1e-4));
            }
          }
          else {  // <---  Next state is whether Surf_Sat or Perch_Sat...
            BB = -dt*(qn - (recharge + cn->getRiNew()*Cos))/(theta_s-theta_r) -
                  epsilon/f*exp(-f*cn->getNtOld()/epsilon);
            AA = -exp(f*(BB-cn->getNtOld())/epsilon);
            cn->setNtNew(cn->getNtOld() + (epsilon/f*LambertW(AA) - BB));
            //  <---- Nt goes down!!!
            if (AA < (-1/exp(1)))
              cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
            cn->setRuNew(ksat*exp(-f*cn->getNtNew()));
          }
        }
      }  // End of TYPE 2

      //first, the unsaturated contribution...

      if ((cn->getNtNew()==cn->getNfNew())&&(cn->getNfNew()!=cn->getNwtNew())
            && (cn->getNfNew() > 10)) {
        cn->setQpout(cn->getNfNew()*cn->getRuNew()*(UAr-1)
        /(ce->getLength()*1000));
      }

      if ((cn->getNfNew()-cn->getNtNew()) > 1) {
        // Perched Sat Contribution
        temphold = cn->getNtNew()*cn->getRuNew()*(UAr-1)+
           ksat*UAr/f*(exp(-f*cn->getNtNew())-exp(-f*cn->getNfNew()));
        temphold -= ksat*f*(cn->getNfNew()-cn->getNtNew())*(cn->getNfNew()
          -cn->getNtNew())/(exp(f*cn->getNfNew())-exp(f*cn->getNtNew()));
        temphold = temphold/(ce->getLength()*1000);
        cn->setQpout(temphold);
      }
      cn->setQpout(cn->getQpout()*Sin);

      } // matches a misindented ELSE early in this case...

    break;

    case Storm_Evol:

      if (cn->getID()==600) // SMR Debug!
        cout << "We are in Storm_Evol!" << endl;

/*  This case was initiated to delete "instable solution"-looking results
    The reason for such a solution is: having been reached by the first
    wetting front, water table gets closer to the surface and, hence, can
    now be reached faster than the GWT of the initial state.
    ---> As a result, when in upper pixels the first wetting front reaches
    water table, in this pixel the second front can reach WT, or third (!)
    can just get started.

    It is assumed that only in pixels with _shallow_ saturated zone wetiing
    front can reach WT during rainfall event ---> every following incoming
    water gets redistributed along the profile <==== physical sense: rise
    of capillary fringe.
```

```
    ***ISSUES to discuss***: 1) for a rainfall event of long duration with
    insignificant rate values we WILL redistribute moisture for dt along
    _deep_ profile and finally when we reach WT, we will rise WT level by
    every succsessive amount of water (until interstorm period starts...)
    Is this reasonable?  ---> CAN be considered as an approximation,
    otherwise we get DISCONTINUITY in the wetting front profile.
    2) When an interstorm period starts, I _do not_ preserve this condition
    anymore ---> any following rainfall would cause formation of wetted edge
    in the soil moisture profile ----> if interstorm period was not long
    enough to destroy un/saturated edges in some uphillslope pixels --->
    I get _discontinuity_ again...
*/

      recharge = (cn->getRain() + cn->getQpin() - cn->getQpout())*Cos;

      AA = (cn->getMiOld() + recharge*dt) - epsilon/f*(theta_s-theta_r) -
           theta_s*cn->getNwtOld();
      BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
      if (BB < (-1/exp(1)))
        cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
      cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
      cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
      cn->setRuNew(0.0);
      cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1 -
           exp(-f*cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew());
      cn->setMuNew(cn->getMiNew());
      cn->setQpout(0.0);

      if (cn->getRain() <= RADAR) {
        cn->addIntStormVar(dt);
        if (cn->getNfOld()==cn->getNwtOld()) {
          if (cn->getIntStormVar() >= IntStormMax) {
            cn->setNfNew(0.0);
            cn->setNtNew(0.0);
          }
          else {
            cn->setNfNew(cn->getNwtNew());
            cn->setNtNew(cn->getNwtNew());
          }
        }
        else {
          cn->setNfNew(0.0);
          cn->setNtNew(0.0);
        }
      }
      else {
        if (cn->getIntStormVar() > 0) cn->setIntStormVar(0.0);
        cn->setNfNew(cn->getNwtNew());
        cn->setNtNew(cn->getNwtNew());
      }

    break;

    case Storm_Unsat_Evol:

      if (cn->getID()==600) // SMR Debug!
        cout << "We are in Storm_Unsat_Evol!" << '\n';

      if (cn->getRain() <= RADAR)
        cn->addIntStormVar(dt);
      else
        cn->setIntStormVar(0.0);

      recharge = (cn->getRain()+cn->getQpout()+cn->getQpin())*Cos;

//      if (cn->getRain()>=Kunsat){
//        cn->sethsrf((cn->getRain()-Kunsat)*Cos);
//        if ((cn->getQpin()-cn->getQpout()) >= 0) {
//          recharge = Kunsat*Cos;
//          cn->setpsrf(cn->getpsrf()+cn->getQpin()-cn->getQpout());
//        }
```

```
//        else
//          recharge = (Kunsat+cn->getQpin()-cn->getQpout())*Cos;
//      }
//      else if (cn->getRain() < Kunsat && (recharge >= Kunsat)) {
//        recharge = Kunsat*Cos;
//        cn->setpsrf(cn->getpsrf()+cn->getQpin()-cn->getQpout());
//      }

// No longer any limits on infiltration in Storm_Unsat_evol!


      qn = recharge;

      cn->setNwtNew(cn->getNwtOld());  // This will not be allowed to change
      cn->setMuNew(cn->getMuOld() + recharge*dt);
      cn->setMiNew(cn->getMiOld());
      //reason for checking both Ri & Mi at each time step
      cn->setRiNew(cn->getRiOld());

      // IIIIIIIIIIIIII  1st TYPE: NO FRONTS EXIST... IIIIIIIIIIIIII

      if (cn->getNfOld()==0 && cn->getNtOld()==0) {
      // start from initialized state...
        ThRiNf=pow((cn->getRiOld()/ksat),(1/epsilon))*(theta_s-theta_r)+theta_r;
        if (cn->getRain() < Kunsat && recharge/Cos < Kunsat) {
          ThReNf = pow(((recharge+Kunsat*Cos)/ksat),(1/epsilon))*
                   (theta_s-theta_r)+theta_r;
          qn = recharge;
          cn->setNfNew(dt*qn/(ThReNf-ThRiNf));
        }
        else {
          if (recharge < ksat) {
            ThReNf = theta_r+(theta_s-theta_r)*pow((recharge/ksat),(1/epsilon));
            SeIn = pow(((ThRiNf-theta_r)/(theta_s-theta_r)),(3 + 1/PoreInd) );
            Se0  = pow(((ThReNf-theta_r)/(theta_s-theta_r)),(3 + 1/PoreInd) );
            G = -Psib/PoreInd*(Se0 - SeIn)/(3 + 1/PoreInd);
            qn = ksat*exp(-f*recharge/(ThReNf-ThRiNf))*G*(ThReNf-ThRiNf)
                 /recharge + recharge;
            //  qn = R1*G*(ThReNf-ThRiNf)/(R1*dt);
            cn->setNfNew(dt*qn/(theta_s-ThRiNf));
          }
          else  {    //  <------  R1 >= Ksat
            ThReNf = theta_s;
            SeIn = pow(((ThRiNf-theta_r)/(theta_s-theta_r)),(3 + 1/PoreInd));
            G = -Psib/PoreInd*(1 - SeIn)/(3 + 1/PoreInd);
            qn = ksat*G*(ThReNf-ThRiNf)/ksat + recharge;
            //  qn = Ksat*G*(ThReNf-ThRiNf)/(R1*dt);
            cn->setNfNew(dt*qn/(ThReNf-ThRiNf));
          }
        }

        if (cn->getNfNew() < 1.0) cn->setNfNew(1.0); //forcing to a deeper value

        cn->setNtNew(cn->getNfNew());

        // calculation of Requivalent:

        Mdelt = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*(1-exp(f*(
          cn->getNfNew()-cn->getNwtNew())/epsilon))+ theta_r*(cn->getNwtNew()-
          cn->getNfNew()));
        AA = epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNfNew()/epsilon))+
          theta_r*cn->getNfNew();

        if (Mdelt >= AA) {
          // ### CASE 1 ###
          if (fabs(Mdelt-AA) <= 1.0e-6) {
            cn->setNtNew(cn->getNfNew());
            cn->setNfNew(cn->getNfNew()+1.0e-5);
            cn->setRuNew(ksat*exp(-f*cn->getNtNew()));
          }
          // ### CASE 2 ###
```

```
          else {
            if (Mdelt >= cn->getNfNew()*theta_s) {
              ThRiNf=theta_r + exp(f*cn->getNfNew()/epsilon)*(theta_s-theta_r)*
                     pow((cn->getRiNew()/ksat),(1/epsilon));
              cn->setNfNew(cn->getNfNew()+(Mdelt-cn->getNfNew()*theta_s)/
                   (theta_s-ThRiNf));

              if (cn->getNfNew() >= cn->getNwtNew()) {
                cn->setNfNew(0.0);
                cn->setNtNew(0.0);
                cn->setNwtNew(0.0);
                cn->setMuNew(0.0);
                cn->setMiNew(0.0);
                cn->setRuNew(0.0);
                cn->setRiNew(0.0);
              }
              else {
                cn->setNtNew(0.0);
                cn->setRuNew(ksat*f*cn->getNfNew()/(exp(f*cn->getNfNew())-1));
                AA=(epsilon/f*(theta_s-theta_r)*(1-exp(f*(cn->getNfNew()-
                  cn->getNwtNew())/epsilon))+theta_r*(cn->getNwtNew()-
                  cn->getNfNew()));
                BB = cn->getNfNew()*theta_s;
                cn->setMuNew(AA+BB);

              }
            }
            else if (Mdelt < cn->getNfNew()*theta_s) {
              BB = (Mdelt-theta_r*cn->getNfNew())/(theta_s-theta_r);
              BB = BB*f/(epsilon*(exp(f*cn->getNfNew()/epsilon)-1));
              cn->setRuNew(ksat*pow(BB,epsilon)); // approximately..
              cn->setNtNew((log(ksat/cn->getRuNew()))/f);
              cn->setNfNew(cn->getNfNew()+1.0e-5);
            }
          }
        }
        else {
          BB = (Mdelt-theta_r*cn->getNfNew())/(theta_s-theta_r);
          BB = BB*f/(epsilon*(exp(f*cn->getNfNew()/epsilon)-1));
          cn->setRuNew(ksat*pow(BB,epsilon)); // approximately...
        }


        if ((cn->getRiNew()-cn->getRuNew()) < 1e-2
               && (cn->getRiNew() - cn->getRuNew()) > 0 ) {
          cn->setRuNew(cn->getRiNew());
        }
        else if ((cn->getRiNew() - cn->getRuNew()) > 1e-2)
          cout << "WARNING!!! UNSAT: RuNew < RiNew: ipx = "<<cn->getID()<< endl;

        if (cn->getRuNew() > ksat)
          cout << "Warning-UNSAT: RuNew > ksat" << endl;

/*

        Nstar = (log(ksat/cn->getRuNew()))/f;
        if (Nstar > cn->getNwtNew())
          cout << "Warning-Nstar > WTdepth < 0" << endl;

        ThRiNf = thetasur;
        ThReNf = theta_r + (theta_s-theta_r)*pow((cn->setRuNew()/ksat),
                 (1/epsilon));
        cn->setNfNew(dt*recharge/(ThReNf-ThRiNf));
        cn->setNtNew(cn->getNfNew());

        // ### Type 1-CASE A ###
        if (fabs(cn->getNfNew()-Nstar) <= 1.0e-3) { //  ~= tolerance value                    cn-
>setNfNew(Nstar + 0.0001); //  Next case will be Perched_Evol..
          cn->setNtNew(Nstar);
          Mdelt = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*
                  (1 - exp(f*(cn->getNtNew()-cn->getNwtNew())/epsilon))
```

```
            + theta_r*(cn->getNwtNew()-cn->getNtNew()));
    BB = (Mdelt - theta_r*cn->getNtNew())/(theta_s-theta_r);
    BB = BB*f/(epsilon*(exp(f*cn->getNtNew()/epsilon) - 1));
    cn->setRuNew(ksat*pow(BB,epsilon));
  }

  // ### Type 1-CASE B ###
  else if (cn->getNfNew() < Nstar) {
    Mdelt = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*
            (1 - exp(f*(cn->getNfNew()-cn->getNwtNew())/epsilon))
            + theta_r*(cn->getNwtNew()-cn->getNfNew()));
    BB = (Mdelt - theta_r*cn->getNfNew())/(theta_s-theta_r);
    BB = BB*f/(epsilon*(exp(f*cn->getNfNew()/epsilon) - 1));
    cn->setRuNew(ksat*pow(BB,epsilon));
  }

  // ### Type 1-CASE C ###
  else if (cn->getNfNew() > Nstar) {
    cn->setNfNew(Nstar);
    Mdelt=cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*
          (1-exp(f*(cn->getNfNew()-cn->getNwtNew())/epsilon)) +
          theta_r*(cn->getNwtNew()-cn->getNfNew()));
    if (Mdelt < cn->getNfNew()*theta_s) {
    // top front still lower than terrain surface...

    //
    // The following method gives some _error_ which increases with the
    // difference:  (Mdelt - Eps/F*(Ths-Thr)*(1 - exp(-F*NfNew/Eps)) +
                                    //                Thr*NfNew)
    // as long as LambertW f_n is iterative, it is more desirable
    // to use this procedure, THOUGH you still can use LambertW f_n...
    //

      AA = epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNfNew()/epsilon))
           + theta_r*cn->getNfNew();
      BB = (Mdelt - theta_r*cn->getNfNew())/(theta_s-theta_r);
      BB = BB*f/(epsilon*(exp(f*cn->getNfNew()/epsilon) - 1));
      cn->setRuNew(ksat*pow(BB,epsilon));

      if (Mdelt > AA) {    //  ...difference is positive..
        cn->setNtNew((log(ksat/cn->getRuNew()))/f);
        //  *** delta = Mdelt-Mi~NfNew => should not be very big value...
        if (cn->getNtNew() > cn->getNfNew()) {
          cn->setNtNew(cn->getNfNew());
          cn->setNfNew(cn->getNfNew()+0.0001);
          cout << "Warning!!! Incorrect estimation of Nt!" << endl;
        }
      }
      else {  // AA > Mdelt... still USat. evol. on the next time step..
        cn->setNtNew(cn->getNfNew());
      }
    }

    else {     // <-- water emerges to the surface...
      cn->setNtNew(0.0);
      cn->setsbsrf(cn->getsbsrf()+(Mdelt - cn->getNfNew()*theta_s)/dt);
      cn->setRuNew(0.0);
      cn->setMuNew(cn->getMuNew() - cn->getsbsrf()*dt);
    }
  }

*/

}
// ------------------------------------------------------------------
// ------------------------------------------------------------------

else if (cn->getNfOld() > 0 && cn->getNtOld() > 0) {
// Let's define Re accounting for old edge and new..
  Mdelt = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*
          (1-exp(f*(cn->getNfOld() - cn->getNwtNew())/epsilon)) +
```

```
            theta_r*(cn->getNwtNew()-cn->getNfOld()));

        if (Mdelt >= cn->getNfOld()*theta_s) {
        // <-- water gets to the surface...

          ThRiNf=theta_r+exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)*pow((
            cn->getRiNew()/ksat),(1/epsilon));
          SeIn = pow(((ThRiNf-theta_r)/(theta_s-theta_r)),(3 + 1/PoreInd));
          G = -Psib/PoreInd*(1 - SeIn)/(3 + 1/PoreInd);
          G = G*(1-exp(-f*cn->getNfOld()))/(f*cn->getNfOld());
          qn = ksat*f*cn->getNfOld()/(exp(f*cn->getNfOld())-1);

          if ((qn*(Cos+G/cn->getNfOld())-cn->getRiOld()*Cos) <= recharge)  {
          //  <--- pass to SurfSatModel
            qn *= (Cos + G/cn->getNfOld());

            if (recharge >= (qn - cn->getRiOld()*Cos)) {
            // Hortonian runoff is generated...
                  cn->sethsrf(cn->gethsrf()+(recharge -(qn-cn->getRiOld()*Cos)));
                  recharge = recharge - cn->gethsrf();
               }
            else
              cout << "WARNING!!! WRONG def.: R < Keqviv-RiOld*Cos: ipx = "
                << cn->getID() << endl;
          cn->setNtNew(0.0);
          cn->setMuNew(cn->getMuOld() + recharge*dt);
          cn->setNfNew(cn->getNfOld() + dt*(qn-cn->getRiNew()*Cos)/
               (theta_s-ThRiNf));
          cn->setRuNew(ksat*f*cn->getNfNew()/(exp(f*cn->getNfNew())-1));

          // Check to see if WT is needed to be modified

          if ((fabs(cn->getNfNew()-cn->getNwtNew())<=1.0e-3) ||
                (cn->getNfNew() > cn->getNwtNew())) {  // ~= tolerance value..
            cn->setNwtNew(0.0);
            cn->setRuNew(0.0);
            cn->setRiNew(0.0);
                  cn->setMiNew(0.0);
            Mperch = cn->getMuNew() - cn->getNwtOld()*theta_s;
                  if (Mperch > 0)     // Logically, it's greater than "0"...
                    cn->setsbsrf(cn->getsbsrf() + Mperch/dt);
                  cn->setMuNew(0.0);
                  cn->setNtNew(0.0);
            cn->setNfNew(0.0);
               }
        }
        else {   // <------ Just keep UNsaturated edge...

          ThReNf = theta_r + exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)*
                     pow((cn->getRuOld()/ksat),(1/epsilon));

// the above equation was altered - could be incorrect. SMR

          Se0  = pow( ((ThReNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
          G = -Psib/PoreInd*(Se0 - SeIn)/(3 + 1/PoreInd);
          G = G*(1-exp(-f*cn->getNfOld()))/(f*cn->getNfOld());
          qn = cn->getRuOld()*Cos+ksat*exp(-f*cn->getNfOld())*G/cn->getNfOld();

          cn->setNfNew(cn->getNfOld()+dt*(qn-cn->getRiNew()*Cos)/
                 (theta_s-ThRiNf));
          cn->setNtNew(cn->getNfNew());

          if (cn->getNfNew() < cn->getNwtNew()) {
            Mdelt = cn->getMuNew()-(epsilon/f*(theta_s -theta_r)*(1-exp(f*
(cn->getNfNew()-cn->getNwtNew())/epsilon))+theta_r*
                     (cn->getNwtNew()-cn->getNfNew()));
            Mdelt = (Mdelt - theta_r*cn->getNfNew())/(theta_s-theta_r);
            Mdelt = Mdelt*f/(epsilon*(exp(f*cn->getNfNew()/epsilon) - 1));
            cn->setRuNew(ksat*pow(Mdelt,epsilon));
          }
```

```
      else {
            if (cn->getMuNew() > theta_s*cn->getNwtNew()) {
         cn->setsbsrf(cn->getsbsrf()+cn->getMuNew()-theta_s
                *cn->getNwtNew());
              cn->setNfNew(0.0);
         cn->setNtNew(0.0);
         cn->setNwtNew(0.0);
         cn->setMuNew(0.0);
         cn->setMiNew(0.0);
         cn->setRuNew(0.0);
         cn->setRiNew(0.0);
            }
      }
   }
}

   else { // ..reservoir in the USat. zone large enough for water...
     Mdelt = (Mdelt - theta_r*cn->getNfOld())/(theta_s-theta_r);
     Mdelt = Mdelt*f/(epsilon*(exp(f*cn->getNfOld()/epsilon) - 1));
     cn->setRuNew(ksat*pow(Mdelt,epsilon));
     Nstar = (log(ksat/cn->getRuNew()))/f;

     if (Nstar > cn->getNwtNew()) {
       Nstar = cn->getNwtNew();
       if (cn->getRuNew() < cn->getRiOld()) {
              cn->setNfNew(0.0);
              cn->setNtNew(0.0);
              cn->setRuNew(0.0);
              cn->setMuNew(cn->getMiNew());
       }
            cout << "WARNING!!! (Nstar > WT depth)" << endl;
     }

     // ###############################################################
     // The following situation usually happens when new water added to
     // moisture content above NfOld exceeds initialization moisture
     // corresponding to NfOld ==>   Mi~ NfOld but still less than
     // maximum moisture capacity ==>   NfOld*Ths
     // *Used here method to redefine Ntop and Ru is _approximate_
     // *More accurate way would be using LambertW function...

     if (Nstar <= cn->getNfOld()) {
       cn->setNtNew(Nstar);
            cn->setNfNew(cn->getNfOld()+1e-5);
     }

     //  <====== well, I also should "move" Wetting front farther, but...
     //  it is inconvenient because this needs rewriting Perched_Sat case
     // SO, with ANOTHER implementation, I have to call "Perched_Evol" f_n

     else {   //  ...NfOld < Nstar...
       ThRiNf = theta_r+exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)*
                 pow((cn->getRiNew()/ksat),(1/epsilon));
       // cout << "ThRiNf : " << ThRiNf << endl;
       ThReNf = theta_r+exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)*
                      pow((cn->getRuOld()/ksat),(1/epsilon));  // last - Mdelt
       // cout << "ThReNf : " << ThReNf << endl;

       SeIn = pow(((ThRiNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
       Se0  = pow(((ThReNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
            G = -Psib/PoreInd*(Se0 - SeIn)/(3 + 1/PoreInd);
       G = G*(1-exp(-f*cn->getNfOld()))/(f*cn->getNfOld());
            qn = cn->getRuOld()*Cos+ksat*exp(-f*cn->getNfOld())
              *G/cn->getNfOld();


            if (qn > 5*recharge) { // FILTERING! NECESSARY AT THE BEGINNING!!!
              qn = 5*recharge;  // <--- NUMERICAL ISSUES!!!
              if (qn <= cn->getRiNew()*Cos || qn <= (cn->getRiNew()*Cos+1) )
                qn = cn->getRuOld()*Cos; // just gravitational component
       }
```

```
        cn->setNfNew(cn->getNfOld()+
             dt*(qn-cn->getRiNew())*Cos/(ThReNf-ThRiNf));
        // cout << "NfNew = " << NfNew << endl;
        cn->setNtNew(cn->getNfNew());

        Mdelt = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*
                (1 - exp(f*(cn->getNfNew()-cn->getNwtNew())/epsilon))+
                 theta_r*(cn->getNwtNew()-cn->getNfNew()));
        BB = (Mdelt - theta_r*cn->getNfNew())/(theta_s-theta_r);
        BB = BB*f/(epsilon*(exp(f*cn->getNfNew()/epsilon) - 1));
        cn->setRuNew(ksat*pow(BB,epsilon));

      }          // matches NfOld < Nstar...
    }
  }

  if (cn->getNfNew() > cn->getNwtNew()) {

  //  NOW, we have to redistribute moisture along the profile:
  //  so that pixel would be the same as in an initialized state...
  //   ...Nwt is rising..

  AA = cn->getMuNew()-epsilon/f*(theta_s-theta_r)-
               theta_s*cn->getNwtOld();
  BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
  if (BB < (-1/exp(1)))
         cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
  cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
  cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
  cn->setRuNew(0.0);
  cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*
            cn->getNwtNew()/epsilon))+theta_r*cn->getNwtNew());
  cn->setMuNew(cn->getMiNew());
  cn->setNfNew(cn->getNwtNew());
  cn->setNtNew(cn->getNwtNew());
  }

  //first, the unsaturated contribution...

  if ((cn->getNtNew()==cn->getNfNew())&&(cn->getNfNew()!=cn->getNwtNew())
          && (cn->getNfNew() > 10)) {
    cn->setQpout(cn->getNfNew()*cn->getRuNew()*(UAr-1)
    /(ce->getLength()*1000));
  }

  if ((cn->getNfNew()-cn->getNtNew()) > 1) {
    // Perched Sat Contribution
    temphold = cn->getNtNew()*cn->getRuNew()*(UAr-1)+
       ksat*UAr/f*(exp(-f*cn->getNtNew())-exp(-f*cn->getNfNew()));
    temphold -= ksat*f*(cn->getNfNew()-cn->getNtNew())*(cn->getNfNew()
      -cn->getNtNew())/(exp(f*cn->getNfNew())-exp(f*cn->getNtNew()));
    temphold = temphold/(ce->getLength()*1000);
    cn->setQpout(temphold);
  }
  cn->setQpout(cn->getQpout()*Sin);

  if (cn->getsbsrf()<0)
    cout << "WARNING!!! RUNOFF component (sbsrf) < 0" << endl;

break;

case Perched_Evol:
  if (cn->getID()==600) // SMR Debug!
    cout << "We are in Perched_Evol!" << '\n';


  //  Perched Saturation has already developed...
  //  *********************************
  //         N E C E S S A R Y:
  //       !!!  recharge > 0  !!!
```

```
//  *********************************

if (cn->getRain() <= RADAR)
  cn->addIntStormVar(dt);
else
  cn->setIntStormVar(0.0);

recharge = (cn->getRain() + cn->getQpin() - cn->getQpout())*Cos;

cn->setNwtNew(cn->getNwtOld());
// This will change ONLY if NfNew hits NwtNew
cn->setMuNew(cn->getMuOld() + recharge*dt);
cn->setMiNew(cn->getMiOld());
cn->setRiNew(cn->getRiOld());

//  ***********************************************************
//  Wet. front is still allowed to move, whatever happens to Nt
//  ***********************************************************

/* ---- Wetting front evolution ---- */

ThRiNf = theta_r+exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)*
         pow((cn->getRiNew()/ksat),(1/epsilon));

SeIn = pow(((ThRiNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
G = -Psib/PoreInd*(1 - SeIn)/(3 + 1/PoreInd);
G = G*(1-exp(-f*cn->getNfOld()))/(f*cn->getNfOld());
qn = ksat*f*(cn->getNfOld()-cn->getNtOld())/
        (exp(f*cn->getNfOld())-exp(f*cn->getNtOld()));
qn *= (Cos + G/cn->getNfOld());

cn->setNfNew(cn->getNfOld()+dt*(qn-cn->getRiNew()*Cos)
             /(theta_s-ThRiNf));


//  ***********************************************************
//     C H E C K  for possible situations with Nf & Nwt....
//  ***********************************************************

if (fabs(cn->getNfNew() - cn->getNwtNew()) <= 1.0e-3) {
  cn->setNwtNew(cn->getNtOld());
  // <=== this will be my flag for Ntop dynamics..
  cn->setNfNew(cn->getNwtNew());
}
else if (cn->getNfNew() > cn->getNwtNew()) {
  cn->setNwtNew(cn->getNtOld());
  // <=== this will be my flag for Ntop dynamics..
  cn->setNfNew(cn->getNwtNew());
  //  *** Now, let's increase inflow by the excess flow... ***
  recharge += qn - ((cn->getNwtOld()-cn->getNfOld())*
               (theta_s-ThRiNf)/dt+cn->getRiNew()*Cos);
}

/* ---- Top front evolution ---- */

//  ***********************************************************
//      <<<<<<<<< NfNew has hit the water table >>>>>>>>>>
//  ***********************************************************

if (cn->getNfNew() == cn->getNwtNew()) {
  if (cn->getMuNew() >= cn->getNwtOld()*theta_s) {
    cn->setNwtNew(0.0);
    cn->setNtNew(0.0);
    cn->setNfNew(0.0);
    cn->setsbsrf((cn->getMuNew()-cn->getNwtOld()*theta_s)/dt);
    cn->setMuNew(0.0);
    cn->setMiNew(0.0);
    cn->setRuNew(0.0);
    cn->setRiNew(0.0);
  }
  else {
```

```
    BB = -dt*(qn-(recharge+cn->getRiNew()*Cos))/(theta_s-theta_r) -
         epsilon/f*exp(-f*cn->getNtOld()/epsilon);
        AA = -exp(f*(BB-cn->getNtOld())/epsilon);
    if (AA < (-1/exp(1))) {  //due to numeric errors...
      cn->setNwtNew(0.0);
      cn->setNtNew(0.0);
      cn->setNfNew(0.0);
      cn->setMuNew(0.0);
      cn->setMiNew(0.0);
      cn->setRuNew(cn->getRiNew());
    }
    else {
      cn->setNtNew(cn->getNtOld()+(epsilon/f*LambertW(AA) - BB));
      cn->setNwtNew(cn->getNtNew());
      cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
      cn->setRuNew(0.0);
      cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f
          *cn->getNwtNew()/epsilon)) + theta_r*cn->getNtNew());
      cn->setMuNew(cn->getMiNew());
      cn->setNtNew(cn->getNwtNew());
      cn->setNfNew(cn->getNwtNew());
    }
  }
}

//  ************************************************************
//    <<<<<<<<< If NfNew is still above water table >>>>>>>>>>
//  ************************************************************

else if (cn->getNfNew() != cn->getNwtNew()) {
  Mdelt = epsilon/f*(theta_s-theta_r)*
          (1-exp(-f*cn->getNtOld()/epsilon))+theta_r*cn->getNtOld();
  Mdva  = epsilon/f*(theta_s-theta_r)*
          (1-exp(-f*cn->getNfNew()/epsilon))+theta_r*cn->getNfNew();
  AA = (Mdelt+(cn->getNfNew()-cn->getNtOld())*theta_s - Mdva);
  if ((recharge+cn->getRiNew()*Cos) < (qn - AA/dt)) {
  // <--- Next state is UnSat.!!!
    Mperch = cn->getMuNew()-(epsilon/f*(theta_s-theta_r)*(1-exp(f*
        (cn->getNfNew()-cn->getNwtNew())/epsilon))+theta_r*
        (cn->getNwtNew()-cn->getNfNew()));
    Mperch = (Mperch - theta_r*cn->getNfNew())/(theta_s-theta_r);
    Mperch = Mperch*f/(epsilon*(exp(f*cn->getNfNew()/epsilon) - 1));
    cn->setRuNew(ksat*pow(Mperch,epsilon));
    //  Equivalent recharge rate above Nf (approx.)
    cn->setNtNew(cn->getNfNew());
  }
  else {    // <---  Next state is whether Surf_Sat or Perch_Sat...
    xxsrf = (recharge+cn->getRiNew()*Cos-qn)*dt + Mdelt;

    if(xxsrf >= cn->getNtOld()*theta_s) {
    //  influx is sufficient to fill space above Ntop
      cn->setNtNew(0.0);    //  wetting front still moves!!!
           cn->sethsrf(cn->gethsrf()+(xxsrf - cn->getNtOld()*theta_s)/dt);
      cn->setMuNew(cn->getMuNew()-cn->gethsrf()*dt);
    }
    else {
      BB = -dt*(qn - (recharge + cn->getRiNew()*Cos))/(theta_s-theta_r) -
           epsilon/f*exp(-f*cn->getNtOld()/epsilon);
      AA = -exp(f*(BB-cn->getNtOld())/epsilon);
      cn->setNtNew(cn->getNtOld() + (epsilon/f*LambertW(AA) - BB));
      if (AA < (-1/exp(1)))
        cout << "WARNING!!! Value for LAMBERT f-n is too small";

      cn->setRuNew(ksat*exp(-f*cn->getNtNew()));
    }

    if (cn->getNwtNew() != 0) {  //only when Nf has not hit the WT..
    // this is an artifical algorithm used to check numerics

    AA = (epsilon/f*(theta_s-theta_r)*(1-exp(f*cn->getNfNew() -
       cn->getNwtNew())/epsilon))+theta_r*(cn->getNwtNew()-cn->getNfNew());
```

```
      BB = (cn->getNfNew()-cn->getNtNew())*theta_s + epsilon/f*(theta_s-
         theta_r)*(1-exp(-f*cn->getNtNew()/epsilon))+theta_r*cn->getNtNew();

      if (cn->getMuNew() > (AA + BB)) {
        ThRiNf = theta_r + exp(f*cn->getNfNew()/epsilon)*(theta_s-theta_r)
          *pow((cn->getRiNew()/ksat),(1/epsilon));
        cn->setNfNew(cn->getNfNew()+(cn->getMuNew()-(AA+BB))/(theta_s -
             ThRiNf));
        if (cn->getNfNew() >= cn->getNwtNew()) {
          if (cn->getNtNew() > 0) {
            cn->setNwtNew(cn->getNtNew());
            cn->setNfNew(cn->getNtNew());
            cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*
              cn->getNwtNew()/epsilon))+theta_r*cn->getNwtNew());
            cn->setMuNew(cn->getMiNew());
            cn->setRiNew(cn->getRuNew());
            cn->setRuNew(0.0);
          }
          else if (cn->getNtNew()==0) {
            cn->setNtNew(0.0);
            cn->setNfNew(0.0);
            cn->setNwtNew(0.0);
            cn->setMuNew(0.0);
            cn->setMiNew(0.0);
            cn->setRiNew(0.0);
            cn->setRuNew(0.0);
          }
        }
        else {
          cn->setMuNew(AA+BB);
          if (cn->getNtNew()==0)
            cn->setRuNew(ksat*f*cn->getNfNew()/(exp(f*cn->getNfNew())-1));
        }
      }
    }
  }
  }

  //  I DO allow to change Nf in the case xxsrf > NtOld*Ths !!

  //first, the unsaturated contribution...

  if ((cn->getNtNew()==cn->getNfNew())&&(cn->getNfNew()!=cn->getNwtNew())
           && (cn->getNfNew() > 10)) {
    cn->setQpout(cn->getNfNew()*cn->getRuNew()*(UAr-1)
    /(ce->getLength()*1000));
  }

  if ((cn->getNfNew()-cn->getNtNew()) > 1) {
    // Perched Sat Contribution
    temphold = cn->getNtNew()*cn->getRuNew()*(UAr-1)+
       ksat*UAr/f*(exp(-f*cn->getNtNew())-exp(-f*cn->getNfNew()));
    temphold -= ksat*f*(cn->getNfNew()-cn->getNtNew())*(cn->getNfNew()
      -cn->getNtNew())/(exp(f*cn->getNfNew())-exp(f*cn->getNtNew()));
    temphold = temphold/(ce->getLength()*1000);
    cn->setQpout(temphold);
  }
  cn->setQpout(cn->getQpout()*Sin);

   if (cn->getsbsrf()<0)
    cout << "WARNING!!! RUNOFF component (sbsrf) < 0" << endl;

break;

case Perched_SurfSat:
  if (cn->getID()==600) // SMR Debug!
    cout << "We are in Perched_SurfSat!" << '\n';

  if (cn->getRain() <= RADAR)
    cn->addIntStormVar(dt);
  else
```

```
  cn->setIntStormVar(0.0);

if (ksat == 0) {
  cn->setMuNew(0.0);
  cn->setMiNew(0.0);
  cn->setRiNew(0.0);
  cn->setRuNew(0.0);
  cn->setNfNew(0.0);
  cn->setNtNew(0.0);
  cn->setNwtNew(cn->getNwtOld());
  cn->setQpout(0.0);
  cn->sethsrf(cn->gethsrf()+cn->getRain()*Cos);
}
else {

// TopFront at Surface, Wetting Front Above WT

  cn->setNwtNew(cn->getNwtOld());
  // Will be allowed to change only if NfNew hits NwtNew

  if ((cn->getNtOld() > 0) && (cn->getNfOld() > 0) &&
       (cn->getNfOld() != cn->getNwtOld())) {
  // (R > Ksat) pixel with devel. UnSat zone.
  // ...(MuOld - QpOut*dt) <-- the amount of water we have to redistribute
  //  ...so that pixel would be the same as in an initialized state...
  //   ...Nwt is rising..

    cout << "WARNING! PerchSurf: NtOld > 0; NfOld > 0" << endl;

    AA = (cn->getMuOld()-cn->getQpout()*dt)-epsilon/f*(theta_s-theta_r)-
         theta_s*cn->getNwtOld();
    cn->setQpout(0.0);
    BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
    if (BB < (-1/exp(1)))
      cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
    cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
    cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
    cn->setRiOld(cn->getRiNew());
    cn->setRuNew(0.0);
    Kunsat = cn->getRiNew();
    cn->setMiNew(epsilon/f*(theta_s-theta_r)*
        (1-exp(-f*cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew());
    cn->setMiOld(cn->getMiNew());
    cn->setMuOld(cn->getMiOld());
    cn->setNtNew(0.0);
    cn->setNtOld(0.0);
    cn->setNfNew(0.0);
    cn->setNfOld(0.0);
  }

  if ((cn->getNtOld() == 0 && cn->getNfOld() == 0) ||
       (cn->getNfOld()==cn->getNwtOld())) {
    cout << "Warning: Perch_surf:NtOld=0; NfOld=0 <-Not good!" << endl;

    if (cn->getNfOld()==cn->getNwtOld()) {
      cn->setNfOld(0.0);
      cn->setNtOld(0.0);
    }
    cn->setRiNew(cn->getRiOld());
    cn->setMiNew(cn->getMiOld());

    // Total rate of influx into a pixel...
    // adding the lateral inflow _above_ Nf...

    if (cn->getRain() >= Kunsat) {
      cn->sethsrf((cn->getRain() - Kunsat)*Cos);
      // <--- rate corresponding to a pixel size
      recharge = (Kunsat + (cn->getQpin()-cn->getQpout()))*Cos;
      // recharge is still > Ksat !
      if ((cn->getQpin()-cn->getQpout()) >= 0) { // positive balance...
        qn = Kunsat*Cos;           // max possible inf. rate
```

```
            cn->sethsrf(cn->gethsrf()+cn->getQpin()-cn->getQpout());
          }
               else
                 qn = recharge;

//            if (qn >= cn->getRiNew()*Cos) {
//            cn->setsbsrf(cn->getsbsrf()+recharge-(qn - cn->getRiNew()*Cos));
//              // return flow
//              recharge -= cn->getsbsrf(); //_real_ rate of infiltration
//            }
//            else
//              cout << "Warning-qn < RiNew * Cos !" << endl;

            Mdelt = 0;
          }
          else if ((cn->getRain()<Kunsat) &&
                   (cn->getRain()+cn->getQpin()-cn->getQpout()) >=Kunsat)  {
          // <-- there is no Hortonian runoff!
                  qn = Kunsat*Cos;
            recharge = (cn->getRain()+cn->getQpin()-cn->getQpout())*Cos;
            cn->setpsrf(cn->getpsrf() + recharge - qn);
            recharge -= cn->getpsrf();
            Mdelt = 0;
          }
          else {
            cout << "Warning:Perch_surf: Haven't figured this out yet!" << endl;
            qn = Kunsat*Cos;
            recharge = (cn->getRain()+cn->getQpin()-cn->getQpout())*Cos;
          }

          cn->setMuNew(cn->getMuOld()+recharge*dt);
          ThRiNf = pow((cn->getRiNew()/ksat),(1/epsilon))*(theta_s-theta_r)+
                       theta_r;
          ThReNf = theta_s; //assumption
          cn->setNfNew(dt*qn/(ThReNf-ThRiNf));
          cn->setNtNew(0.0);

        }

        else if ((cn->getNtOld()==0) && (cn->getNfOld()>0) &&
                  (cn->getNfOld()!=cn->getNwtOld())) {
        //surface sat has already developed...

          ThRiNf = theta_r+exp(f*cn->getNfOld()/epsilon)*(theta_s-theta_r)
                     *pow((cn->getRiOld()/ksat),(1/epsilon));
          SeIn = pow(((ThRiNf-theta_r)/(theta_s-theta_r)), (3 + 1/PoreInd) );
          G = -Psib/PoreInd*(1 - SeIn)/(3 + 1/PoreInd);
          G = G*(1-exp(-f*cn->getNfOld()))/(f*cn->getNfOld());
          qn = ksat*f*cn->getNfOld()/(exp(f*cn->getNfOld())-1);
          qn *= (Cos + G/cn->getNfOld());

       // <<< AGAIN, the newcoming moisture is not taken into account >>>
       // <<< I define the flux based on the old state variables. The >>>
       // <<< though is obvious: I can not ADD water unless I know the>>>
       // <<< flux in the soil.....................................>>>

          recharge = (cn->getRain()+(cn->getQpin()-cn->getQpout()))*Cos;
          // Proj. of rain. r. to a _real_ sloped domain

          if (recharge >= (qn - cn->getRiOld()*Cos)) {
          // Hortonian runoff is generated...
                  cn->sethsrf(cn->gethsrf()+ (recharge - (qn - cn->getRiOld()*Cos)));
                  recharge = recharge - cn->gethsrf();
          }

          else
            cout << "WARNING! WRONG state definition: recharge < xxsrf" << endl;
          cn->setNtNew(0.0);
          cn->setRiNew(cn->getRiOld());
          cn->setMiNew(cn->getMiOld());
          cn->setMuNew(cn->getMuOld() + recharge*dt);
```

```
        cn->setNfNew(cn->getNfOld() + dt*(qn-cn->getRiNew()*Cos)
                     /(theta_s-ThRiNf));
      }

      if (cn->getNfNew() >= 1.0) {  // Goes to infinity if not...
        cn->setRuNew(ksat*f*cn->getNfNew()/(exp(f*cn->getNfNew())-1));
        if (cn->getRuNew() > ksat)
          cn->setRuNew(ksat);
      }
      else
        cn->setRuNew(ksat);

      //  Check to see if WT is needed to be modified
       if ((fabs(cn->getNfNew()-cn->getNwtNew()) <= 1.0e-3) ||
          (cn->getNfNew()>cn->getNwtNew())) {
        cn->setNwtNew(0.0);
        cn->setRiNew(0.0);
        cn->setRuNew(0.0);
        cn->setMiNew(0.0);
        Mperch = cn->getMuNew() - cn->getNwtOld()*theta_s;
        if (Mperch > 0)     // Logically, it's greater than "0"...
          cn->setsbsrf(cn->getsbsrf()+ Mperch/dt);
        cn->setMuNew(0.0);
        cn->setNfNew(0.0);
        cn->setNtNew(0.0);
      }

      if (cn->getNwtNew() != 0 && cn->getNfNew() >= 1.0) {
        // this is an artifical algorithm used to check numerics
        AA = (epsilon/f*(theta_s-theta_r)*(1-exp(f*cn->getNfNew() -
           cn->getNwtNew())/epsilon))+theta_r*(cn->getNwtNew()-cn->getNfNew());
        BB = cn->getNfNew()*theta_s;

        if (cn->getMuNew() > (AA + BB)) {
          ThRiNf = theta_r + exp(f*cn->getNfNew()/epsilon)*(theta_s-theta_r)
            *pow((cn->getRiNew()/ksat),(1/epsilon));
          cn->setNfNew(cn->getNfNew()+(cn->getMuNew()-(AA+BB))/(theta_s -
                ThRiNf));
          if (cn->getNfNew() >= cn->getNwtNew()) {
            cn->setNwtNew(0.0);
            cn->setRiNew(0.0);
            cn->setRuNew(0.0);
            cn->setMiNew(0.0);
            cn->setMuNew(0.0);
            cn->setNfNew(0.0);
            cn->setNtNew(0.0);
          }
          else {
            cn->setMuNew(AA + BB);
            cn->setRuNew(ksat*f*cn->getNfNew()/(exp(f*cn->getNfNew())-1)); //keq
          }
        }
      }

      if ((cn->getNfNew()-cn->getNtNew()) > 10) {
    cn->setQpout((ksat*UAr/f*(1-exp(-f*cn->getNfNew()))-ksat*f*cn->getNfNew()          *cn-
>getNfNew()/(exp(f*cn->getNfNew())-1))/(ce->getLength()*1000));
      }
      cn->setQpout(cn->getQpout()*Sin);
      // End of QpOut calculation
    }   //<====== matches ELSE (Ksat != 0)

     if (cn->gethsrf()<0)
      cout << "WARNING!!! RUNOFF component (hsrf) < 0" << endl;
    if (cn->getsbsrf()<0)
      cout << "WARNING!!! RUNOFF component (sbsrf) < 0" << endl;
    if (cn->getpsrf()<0)
      cout << "WARNING!!! RUNOFF component (psrf) < 0" << endl;

  break;
```

```
  } //end of long case definitions

  dnode = (tCNode *)ce->getDestinationPtrNC();
  dnode->addQpin(cn->getQpout());

// Setting the runoff back to the correct stuff for mass balance...

  cn->setsbsrf(cn->getsbsrf()/Cos);
  cn->setpsrf(cn->getpsrf()/Cos);
  cn->sethsrf(cn->gethsrf()/Cos);
  cn->setsrf(cn->getsrf()/Cos);
  cn->setesrf(cn->getesrf()/Cos);
  cn->setrsrf(cn->getrsrf()/Cos);
  cn->setsatsrf(cn->getsatsrf()/Cos);
  cn->setsrf(cn->getsbsrf()+cn->getpsrf()+cn->gethsrf());

  if ((cn->getID() > 598) && (cn->getID() < 603)) { // SMR Debug!
    cout << "For Node #" << cn->getID() << '(' << cn->getX() << ',' <<
         cn->getY() << ')' << '\n';
    cout << "------------------------" <<'\n';
    cout << "Pixel State = " << Pixel_State << '\n';
    cout << "NwtNew = " << cn->getNwtNew() << '\n';
    cout << "MiNew  = " << cn->getMiNew() << '\n';
    cout << "RiOld  = " << cn->getRiOld() << '\n';
    cout << "NfNew  = " << cn->getNfNew() << '\n';
    cout << "NtNew = " << cn->getNtNew() << '\n';
    cout << "Qpin   = " << cn->getQpin()  << '\n';
    cout << "Qpout = " << cn->getQpout() << '\n';
    cout << '\n';
  }

  cn = nodIter.NextP();

  } //end of long while node loop!

} //end of Auroop function

void Cbsim::Reset()
{
  tCNode * cn;
  tMeshListIter<tCNode> nodIter( gridPtr->getNodeList());

  for(cn=nodIter.FirstP(); !(nodIter.AtEnd()); cn=nodIter.NextP())
  {
    cn->setNwtOld(cn->getNwtNew());
    cn->setMuOld(cn->getMuNew());
    cn->setMiOld(cn->getMiNew());
    cn->setNtOld(cn->getNtNew());
    cn->setNfOld(cn->getNfNew());
    cn->setRuOld(cn->getRuNew());
    cn->setRiOld(cn->getRiNew());
    cn->setNwtNew(0.0);
    cn->setMuNew(0.0);
    cn->setMiNew(0.0);
    cn->setNtNew(0.0);
    cn->setNfNew(0.0);
    cn->setRuNew(0.0);
    cn->setRiNew(0.0);
    cn->setQpin(0.0);
    cn->sethsrf(0.0);
    cn->setpsrf(0.0);
    cn->setsbsrf(0.0);
  }
} // end of Reset function

void Cbsim::InitSet()
{
  tCNode * cn;
  tMeshListIter<tCNode> nodIter( gridPtr->getNodeList());
  double midstep;
```

```
  tArray<double> gwaterval(gridPtr->getNodeList()->getSize());
  ifstream gwinfile;
  char oneline[ 120] ;
  char *tokenPtr;
  int firsttime, idno, nodeid;

  for(cn=nodIter.FirstP(); !(nodIter.AtEnd()); cn=nodIter.NextP())
  {
    // still need to set gw level here-keeps the bounds from = 0

    cn->setNwtOld(cn->getZ()*1); //<--- This can be given as input.
    cn->setNwtNew(cn->getNwtOld()); //this helps around edges...
    cn->setRiOld(ksat*exp(-f*cn->getNwtOld()));
    midstep = (1-exp(-f/epsilon*cn->getNwtOld()))*(theta_s-theta_r)*(epsilon/f)
              + theta_r*cn->getNwtOld();
    cn->setMuOld(midstep);
    cn->setMiOld(midstep);
    cn->setQpin(0.0);
//  only really used when setting up the original voronoi areas
// a good just in case measure...
   cn->setRain(0.0);

  }

  gwinfile.open(gwatfile);

  gwinfile >> oneline; //junks the header

  while (gwinfile >> oneline){

    //cout << oneline << endl;

         tokenPtr = strtok( oneline,",");

    firsttime = 0;
         while (tokenPtr != NULL){
      if (firsttime == 0) {
        idno = atoi(tokenPtr);
      }
                    else {
        //cout << "Yep, we are getting here..." << endl;
        gwaterval[ idno] = atof(tokenPtr);
      }

      //cout << "The rain value is..." << rain[ idno] << endl;
      //cout << "The idno is..." << idno << endl;
      firsttime++;
      tokenPtr = strtok( NULL, ",");

    }
  }

  cn = nodIter.FirstP();

  while( nodIter.IsActive())
  {
    nodeid = cn->getID();
    cn->setNwtOld(gwaterval[ nodeid] );
    cn->setNwtNew(cn->getNwtOld());
    cn->setRiOld(ksat*exp(-f*cn->getNwtOld()));
    midstep = (1-exp(-f/epsilon*cn->getNwtOld()))*(theta_s-theta_r)*(epsilon/f)
              + theta_r*cn->getNwtOld();
    cn->setMuOld(midstep);
    cn->setMiOld(midstep);

    cn = nodIter.NextP();
  }

} //end of InitSet function

void Cbsim::GndWater(double dt)
```

```
{
  tCNode * cn;
  tCNode * cnorg;
  tCNode * cndest;
  tEdge * ce;
  double gwslope;
  double volout,  // Water volume output from a node (neg=input)
         dtmax,   // Max global step size (initially equal to total time rt)
         deficit;      // Average Deficit between two edges...
  tMeshListIter<tCNode> nodIter( gridPtr->getNodeList() );
  tMeshListIter<tEdge> edgIter( gridPtr->getEdgeList() );
  dtmax = dt;  // Initialize dtmax to total time step, rt

  for( cn=nodIter.FirstP(); !(nodIter.AtEnd()); cn=nodIter.NextP() )
    cn->setGwaterChng( 0 );

  // Compute sediment volume transfer along each edge

  for( ce=edgIter.FirstP(); edgIter.IsActive(); ce=edgIter.NextP() ) {

  // The equation below is very close to being right.
       // smr - 10/28/98
    cnorg = (tCNode *)ce->getOriginPtrNC();
    deficit = cnorg->getNwtOld();
    cndest = (tCNode *)ce->getDestinationPtrNC();
    deficit += cndest->getNwtOld(); //used to be += to add them together
    deficit = deficit/2;

    gwslope = (cnorg->getZ()-(cnorg->getNwtOld()/1000)-(cndest->getZ()
                -(cndest->getNwtOld()/1000)))/ce->getLength();
    //This has been changed to act as mm^3 values!
    volout = ((ksat*gwslope*ce->getVEdgLen()*1000)/f)*dtmax
             * exp(-f*deficit); //for average val, divide by 2

    // Record outgoing flux from origin
    cn = (tCNode *)ce->getOriginPtrNC();
    cn->addGwaterChng( volout );
    // Record incoming flux to dest'n
    cn = (tCNode *)ce->getDestinationPtrNC();
    cn->addGwaterChng( -volout );

    if ((cnorg->getID()==16) || (cndest->getID()==16)) {  // SMR debug
      cout << '\n' << "Saturated Area:" << '\n';
      cout << volout << " mass exch. from " << ce->getOriginPtr()->getID()
           << " to " << ce->getDestinationPtr()->getID()
           << " on slp " << gwslope << " ve " << ce->getVEdgLen()
           << "deficit" << deficit<< endl;
      cout << "Orginator elevation " << cnorg->getZ() << " wtelev "
           << cnorg->getNwtOld() << endl;
      cout << "Destination elevation " << cndest->getZ() << " wtelev "
           << cndest->getNwtOld() << endl;
    }

    ce = edgIter.NextP();  // Skip complementary edge
  }

  // Compute loss/gain at water table for each node
  for( cn=nodIter.FirstP(); nodIter.IsActive(); cn=nodIter.NextP() ) {
    cn->setNwtNew(cn->getNwtOld()
            +(cn->getGwaterChng()*.000001/cn->getVArea()));

    // the previous statement adds or subtracts net flux/area
    //cout << cn->getGwaterChng() << " VA " << cn->getVArea() << endl;

  } // End GW equation (transmissivity type approach)

  // Coupling with Unsat+Perched Zone moisture from mass balance considerations
  // Again, separate out into cases & handle each through a switch statement

  double tempgw, alpha, Cos, Nstar;
  int Couple_State;
```

```
double Mdelt, dM1, dM2, dM3, AA, BB; //added to handle Valeri's Code...

enum { GW_Exfiltrate,GW_IntStorm_Like,GW_Initial,GW_Positive_Bal};

for( cn=nodIter.FirstP(); nodIter.IsActive(); cn=nodIter.NextP() ) {
  Couple_State = -1000;
  ce = cn->getFlowEdg();
  alpha = atan(ce->getSlope());
  Cos = cos(alpha);

  if (cn->getNwtNew() <= 0) {
    cn->setsatsrf(fabs(cn->getNwtNew()*theta_s)/dt);
    cn->setNwtNew(0.0);
  }
  else
    cn->setsatsrf(0.0);

  //This is where I need to add Valeri's Changes...

  if (cn->getNwtNew()==cn->getNwtOld()) Couple_State=GW_Initial;
  else if (cn->getNwtNew() >  cn->getNwtOld()) Couple_State=GW_IntStorm_Like;
  else if (cn->getNwtNew() <  cn->getNwtOld()) Couple_State=GW_Positive_Bal;
  if (cn->getMuOld() > cn->getNwtNew()*theta_s) Couple_State=GW_Exfiltrate;

  switch (Couple_State) {

    case GW_Exfiltrate:
      if (cn->getID()==16) // SMR Debug!
        cout << "We are in GW_Exfiltrate" << endl;
      cn->setsatsrf(cn->getsatsrf() + (cn->getMuOld() - cn->getNwtNew()
                    *theta_s)/dt);
      cn->setNwtNew(0.0);
      cn->setMuNew(0.0);
      cn->setMiNew(0.0);
      cn->setRiNew(0.0);
      cn->setRuNew(0.0);
      cn->setNfNew(0.0);
      cn->setNtNew(0.0);
    break;

    case GW_Initial:
      if (cn->getID()==16) // SMR Debug!
        cout << "We are in GW_Initial" << endl;
      cn->setMuNew(cn->getMuOld());
      cn->setMiNew(cn->getMiOld());
      cn->setNfNew(cn->getNfOld());
      cn->setNtNew(cn->getNtOld());
      cn->setRuNew(cn->getRuOld());
      cn->setRiNew(cn->getRiOld());
    break;

    case GW_IntStorm_Like:
      if (cn->getID()==16) // SMR Debug!
        cout << "We are in GW_IntStorm_Like" << endl;
      Mdelt = (cn->getNwtNew() - cn->getNwtOld())*theta_s;
      AA = cn->getMiOld() - Mdelt - epsilon/f*(theta_s-theta_r) -
             theta_s*cn->getNwtOld();
      BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
      if (BB < (-1/exp(1)))
        cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
      cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
      cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
      cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1 - exp(-f*cn->getNwtNew()
          /epsilon)) + theta_r*cn->getNwtNew());

      if ((cn->getNfOld() == 0) || (cn->getNfOld()==cn->getNwtOld())) {
        cn->setRuNew(0.0);
        cn->setMuNew(cn->getMiNew());
        cn->setNtNew(0.0);
        if ((cn->getNfOld()==cn->getNwtOld()) && (cn->getNwtOld() != 0))
          cn->setNfNew(cn->getNwtNew());
```

```
    else
      cn->setNfNew(0.0);
  }
  else {   //  <---- There was an edge....
    dM1 = epsilon/f*(theta_s-theta_r)*exp(-f*cn->getNwtNew()/epsilon)
        *(exp(f*cn->getNfOld()/epsilon) - 1) + theta_r*cn->getNfOld();
    dM2 = cn->getMuOld() - cn->getMiOld();
    cn->setMuNew(cn->getMiNew() + dM2);
    // <------ Total moisture content above NwtNew
    Mdelt = dM1 + dM2;
    dM3 = epsilon/f*(theta_s-theta_r)*(1 - exp(-f*cn->getNfOld()/epsilon))
          + theta_r*cn->getNfOld();
    if (Mdelt < dM3) {
      cn->setNfNew(cn->getNfOld());
      cn->setNtNew(cn->getNfNew());
      //Wetting fnt is in the same pos. but the edge has gotten "thinner"
      Mdelt = (Mdelt - theta_r*cn->getNfOld())/(theta_s-theta_r);
      Mdelt = Mdelt*f/(epsilon*(exp(f*cn->getNfOld()/epsilon) - 1));
      cn->setRuNew(ksat*pow(Mdelt,epsilon));
      if (cn->getRuNew() <= cn->getRiNew())
        cout << "Warning: GW Dynamics are wrong. RuNew <= RiNew" << endl;
    }
    else {   //  <---- Perch_Saturated edge OR .... ???
      Mdelt = epsilon/f*(theta_s-theta_r)*exp(-f*cn->getNwtOld()
              /epsilon)*(exp(f*cn->getNfOld()/epsilon) - 1) +theta_r*
              cn->getNfOld()- dM1;
      //the amount of water we have to subtract from the edge....

      if ((cn->getNtOld() > 0) && (cn->getNtOld() < cn->getNfOld())) {
      // There was a Perch_Sat. edge
        dM3 = epsilon/f*(theta_s-theta_r)*(1 - exp(-f*cn->getNtOld()
              /epsilon)) + theta_r*cn->getNtOld();
        AA = dM3-Mdelt-epsilon/f*(theta_s-theta_r) -
             theta_s*cn->getNtOld();
        BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
        cn->setNtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
      }
      else if (cn->getNtOld() == 0 && cn->getNfOld() > 0) {
        AA = -Mdelt;
        BB = -exp(f*AA/(epsilon*(theta_s-theta_r)) - 1);
        cn->setNtNew(epsilon/f*(1 + LambertW(BB)) - AA/(theta_s-theta_r));
      }

      if (BB < (-1/exp(1)))
        cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
      if (cn->getNtNew() > cn->getNfOld())
        cout << "WARNING!!! Wrong Case Defining: NtNew > NfOld" << endl;
      cn->setNfNew(cn->getNfOld());
         cn->setRuNew(ksat*exp(-f*cn->getNtNew()));
    }
  }
break;

case GW_Positive_Bal:
  if (cn->getID()==16) // SMR Debug!
    cout << "We are in GW_Positive_Bal" << endl;

  if ((cn->getNwtNew() > cn->getNfOld()) ||
      (cn->getNfOld()==cn->getNwtOld())) {
  // <---- it has not hit the wetting front
    cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNwtNew()
                 /epsilon)) + theta_r*cn->getNwtNew());
    dM1 = epsilon/f*(theta_s-theta_r)*(1-exp(f*(cn->getNwtNew()
    -cn->getNwtOld())/epsilon))+theta_r*(cn->getNwtOld()-cn->getNwtNew());
    dM2 = cn->getMiOld() - dM1;
    Mdelt = dM1 - (cn->getMiNew()-dM2);
    if (Mdelt<0) cout << "WARNING!!! Mdelt < 0 " << endl;

    // <<<<<<<< NOW, THERE ARE TWO CASES WE NEED TO HANDLE... >>>>>>>>

    if ((cn->getNfOld() == 0)||(cn->getNfOld()==cn->getNwtOld())) {
```

```
        // <----- The 1st one
          AA = (cn->getMiNew() + Mdelt) - epsilon/f*(theta_s-theta_r) -
                 theta_s*cn->getNwtNew();
          BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
          if (BB < (-1/exp(1)))
            cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
          cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
          cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
          cn->setRuNew(0.0);
          cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*cn->getNwtNew()
                       /epsilon)) + theta_r*cn->getNwtNew());
          cn->setMuNew(cn->getMiNew());
          cn->setNtNew(0.0);
          if (cn->getNfOld()==cn->getNwtOld())
            cn->setNfNew(cn->getNwtNew());
          else
            cn->setNfNew(0.0);
        }

        else if ((cn->getNfOld() > 0)&&(cn->getNfOld() != cn->getNwtOld())) {
        // <----- the 2nd one
          cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
          if (Mdelt>10)
            cout << "WARNING!!! Mdelt > 10mm: reconsider algorithm " << endl;
          cn->setMuNew(cn->getMuOld() + Mdelt);
          if (cn->getMuNew() < cn->getMiNew())
            cout << "WARNING! Total moisture content, MuNew < MiNew " << endl;
          Mdelt = cn->getMuNew() - (epsilon/f*(theta_s-theta_r)*(1 - exp(f*
                  (cn->getNfOld() - cn->getNwtNew())/epsilon)) +
                   theta_r*(cn->getNwtNew()-cn->getNfOld()));
          //====> Getting RuNew & defining current pixel state...
          if (Mdelt >= cn->getNfOld()*theta_s) {
          // water gets to the surface... THIS IS NEGOTIABLE...
            cn->setNfNew(cn->getNfOld());
            cn->setNtNew(0.0);
            cn->setsatsrf(cn->getsatsrf() + (Mdelt - cn->getNfNew()
                          *theta_s)/dt);
            cn->setRuNew(0.0);
            cn->setMuNew(cn->getMuNew() - cn->getsatsrf()*dt);
          }
          else {
            if (cn->getNtOld() < cn->getNfOld()) {
            // <--- Pixel is in Perch_Saturated state
            // THIS ALGORITHM IS KIND OF DANGEROUS:
            // <== PRODUCES SIGNIFICANT ERROR IF Mdelt >> Mi ~ NtOld
            // Yep-It is real dangerous! See below.
              Mdelt -= (cn->getNfOld()-cn->getNtOld())*theta_s;
              Mdelt = (Mdelt - theta_r*cn->getNtOld())/(theta_s-theta_r);
              Mdelt = Mdelt*f/(epsilon*(exp(f*cn->getNtOld()/epsilon) - 1));
              cn->setRuNew(ksat*pow(Mdelt,epsilon));
              Nstar = (log(ksat/cn->getRuNew()))/f;
// the following if statement was added by SMR on 4/8/99 as we have a problem
// Nstar = -inf at times, causing NtNew to go bonkers.
              if (Nstar < -1000)
                cn->setNtNew(0.0);
              else
                cn->setNtNew(Nstar);
// end of my hack to keep the model running-needs to be investigated after
// conference.
              cn->setNfNew(cn->getNfOld());
            }
            else if (cn->getNtOld() == cn->getNfOld()) {
              cn->setNfNew(cn->getNfOld());
              cn->setNtNew(cn->getNtOld());
              Mdelt = (Mdelt - theta_r*cn->getNfOld())/(theta_s-theta_r);
              Mdelt = Mdelt*f/(epsilon*(exp(f*cn->getNfOld()/epsilon) - 1));
              cn->setRuNew(ksat*pow(Mdelt,epsilon));
              Nstar = (log(ksat/cn->getRuNew()))/f;

              if (Nstar > cn->getNwtNew()) {
                     Nstar = cn->getNwtNew();
```

```
          if (cn->getRuNew() < cn->getRiOld()) {
                cn->setNfNew(0.0);
            cn->setNtNew(0.0);
            cn->setRuNew(0.0);
            cn->setMuNew(cn->getMiNew());
          }
          cout << "WARNING!!! (Nstar > WT depth)" << endl;
        }

      // ###########################################################
      // The following situation usually happens when water is added to
      // moisture content above NfOld exceeds initialization moisture
      // corresponding to NfOld ==>   Mi~ NfOld but still less than
      // maximum moisture capacity ==>   NfOld*Ths
      // *Used here method to redefine Ntop and Ru is _approximate_
      // *More accurate way would be using LambertW function...

        if (Nstar <= cn->getNfOld()) {
          cn->setNtNew(Nstar);
          // this set of NtNew is OK.
          cn->setNfNew(cn->getNfOld());
        }
      }
    }
  }    // <====== corresponds to else if NfOld > 0
}

       else if ((cn->getNwtNew() <= cn->getNfOld()) && (cn->getNfOld() !=
           cn->getNwtOld())){
// it has struck the wetting front
  if (cn->getNtOld() == cn->getNfOld()) {   // <--- UNSaturated edge..
    cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f
        *cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew());
    // Mi corresponding to NwtNew
    Mdelt = cn->getMuOld()-cn->getMiNew();
    // Mdelt = (MuOld - dM1) - (MiNew - dM1);
    // Mdelt - is the amt of water we have to redistribute above NwtNew
    // it represents an amt of water "swallowed" by rising water table
  }
  else if (cn->getNtOld() < cn->getNfOld()) {
    if (cn->getNwtNew() > cn->getNtOld()) {   // <---- lower top front...
      cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f
          *cn->getNtOld()/epsilon)) + theta_r*cn->getNtOld());
      // Mi correspond. to NtOld
      Mdelt = cn->getMuOld() - (cn->getMiNew()+(cn->getNwtNew()
              -cn->getNtOld())*theta_s);
      cn->setNwtNew(cn->getNtOld());
    }
    else if (cn->getNwtNew() <= cn->getNtOld()) {
    // above top front...
      cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*
          cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew());
      // <-- Mi corresponding to NwtNew
      Mdelt = cn->getMuOld() - cn->getMiNew();  //  May be negative!!!
    }
  }

  if (Mdelt<0) cout << "WARNING!!! Mdelt < 0 " << endl;

  AA = (cn->getMiNew() + Mdelt) - epsilon/f*(theta_s-theta_r) -
       theta_s*cn->getNwtNew();
  BB = -exp(f*AA/(epsilon*(theta_s-theta_r)));
  if (BB < (-1/exp(1)))
    cout << "WARNING!!! Value for LAMBERT f-n is too small" << endl;
  cn->setNwtNew(epsilon/f*LambertW(BB) - AA/(theta_s-theta_r));
  cn->setRiNew(ksat*exp(-f*cn->getNwtNew()));
  cn->setRuNew(0.0);
  cn->setMiNew(epsilon/f*(theta_s-theta_r)*(1-exp(-f*
      cn->getNwtNew()/epsilon)) + theta_r*cn->getNwtNew());
  cn->setMuNew(cn->getMiNew());
  cn->setNtNew(0.0);
```

```
          cn->setNfNew(cn->getNwtNew());
        }
      break;

    } //end of the cases

    if (cn->getsatsrf()<0)
      cout << "WARNING!!! RUNOFF component (satsrf) < 0 " << endl;

    if (cn->getMuNew()<cn->getMiNew())
      cout << "WARNING!!! Total moisture content (MuNew) < MiNew " << endl;

    if (cn->getID() < 20) { // SMR Debug!
      cout << "After Gwater..." << cn->getID() << '(' << cn->getX() << ',' <<                cn-
>getY() << ')' << '\n';
      cout << "-------------------------" <<'\n';
      cout << "NwtNew = " << cn->getNwtNew() << '\n';
      cout << "MuNew  = " << cn->getMuNew() << '\n';
      cout << "MuOld  = " << cn->getMuOld() << '\n';
      cout << "NfNew  = " << cn->getNfNew() << '\n';
      cout << "NtNew = " << cn->getNtNew() << '\n';
      cout << "Qpin   = " << cn->getQpin()  << '\n';
      cout << "Qpout = " << cn->getQpout() << '\n';
      cout << '\n';
    }

  } // end of long for loop

} // end of GndWater Routine...

// ---------------------------------------------------------------
// usage: W(z) or W(n,z)
//
// Compute the Lambert W function of z.  This function satisfies
// W(z).*exp(W(z)) = z, and can thus be used to express solutions
// of transcendental equations involving exponentials or logarithms.
//
// n must be integer, and specifies the branch of W to be computed;
// W(z) is a shorthand for W(0,z), the principal branch.  Branches
// 0 and -1 are the only ones that can take on non-complex values.
//
// If either n or z are non-scalar, the function is mapped to each
// element; both may be non-scalar provided their dimensions agree.
//
// This implementation should return values within 2.5*eps of its
// counterpart in Maple V, release 3 or later.  Please report any
// discrepancies to the author, Nici Schraudolph <nic@idsia.ch>.
// *** Modified by Valeri Ivanov as a C++ implementation code
// *** 08.19.1999
// For further details, see:
//
// Corless, Gonnet, Hare, Jeffrey, and Knuth (1996), "On the Lambert
// W Function", Advances in Computational Mathematics 5(4):329-359.
// ---------------------------------------------------------------


double Cbsim::LambertW(double z)
{
   fcomplex w, v, tt, ttt, t, p;
   double xx, xy;
   double c, f;
   int n;

   if (z == 0)      //  LambertW function in 0 is 0
      return 0;

        // series expansion about -1/e
        //
        // p = (1 - 2*abs(b))*sqrt(2*exp(1)*z + 2);
        // w = (11/72)*p;
        // w = (w - 1/3)*p;
```

```
        // w = (w + 1)*p - 1
        //
        // first-order version suffices:
        //
        //  w = sqrt((2*exp(1)*z + 2)) - 1;

    xx = 2*exp(1)*z + 2;

    if (xx < 0) {
        xx = fabs(xx);
        xx = sqrt(xx);
        setComplex(&w, -1, xx);
    }
    else {
         xx = sqrt(xx) - 1;
         setComplex(&w, xx, 0);
    }

// ***********************************
// assymptotic expansion at 0 and Inf
//    v = log(z);
//    if (v != 0)
//    v = v - log(v);

    xx = fabs(z);
    xx = log(xx);
    if (z < 0)
        setComplex(&v, xx, PI);
    else
        setComplex(&v, xx, 0);

    //    cout << "v = First operation.. ";
    //    Print(&v);

// &&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&

    if (z > 1) {
        xy = log(xx);
        setComplex(&tt, xy, 0);
        setCompNumber(&v, (Csub(v,tt)));
    }
    else if (z == 1)
        setComplex(&tt, 0, 0);
    else if ( z < 1 && z > 0) {
        xy = log(fabs(xx));
        setComplex(&tt, xy, PI);
        setCompNumber(&v, (Csub(v,tt)));
    }
    else if ( z < 0 && z > -1) {

    //  cout << "FF " << v.r << "  SS " << v.i << endl;
        xx = atan(v.i/ v.r);          //     - negative value..
    //  cout << "XX = " << xx << endl;
        xy = log(fabs(v.r / cos(xx)));
        xx += PI;                     //     - positive imaginery part
        setComplex(&tt, xy, xx);
        setCompNumber(&v, (Csub(v,tt)));
    }
    //    cout << "v - Second operation.. ";
    //    Print(&v);

// ####################################

// choose strategy for initial guess
//  c = c | (b*imag(z) > 0) | (~imag(z) & (b == 1));

    c = fabs(z + 1/exp(1));


    if (c > 1.45) {
        c = 1;
```

```
        setCompNumber(&w, v);
//      setCompNumber(&w, (Cadd(RCmul((1 - c),w), RCmul(c,v))));
    }
    else
        c = 0;   // w is the same..

//    Print(&w);
//    w = (1 - c)*w + c*v;
//  ###################################
//  Halley iteration

    n = 0;
    setComplex(&t, 1000, 1000);

    while ((n < 11) && (fabs(t.r) > xy) || (fabs(t.i) > xx)) {

        xx = exp(Cr(&w));
        setComplex(&p, (xx*cos(Ci(&w))), (xx*sin(Ci(&w))));

        //      cout << "\t\t";
        //       Print(&p);

        setCompNumber(&t, (Cmul(w, p)));
        subtReal(&t, z);

        //        cout << "\n\n\nt = "; Print(&t);
        //     p = exp(w);
        //     t = w*p - z;

        if (w.i == 0 && w.r == -1)
           f = 0;
        else
           f = 1;

        setCompNumber(&tt, Cmul(p, CaddFF(&w, f)));

        setCompNumber(&ttt,   Csub(tt,   Cdiv(Cmul(t, RCmul(0.5, CaddFF(&w, 2.0))),   CaddFF(&w,
f))));
        //        cout << "TTT = "; Print(&ttt);

        setCompNumber(&t, Cdiv(RCmul(f,t), ttt));
        //        t = f*t/(p*(w + f) - 0.5*(w + 2.0)*t/(w + f));


        setCompNumber(&w, Csub(w, t));

        xy = (2.48*EPS)*(1.0+fabs(w.r));
        xx = (2.48*EPS)*(1.0+fabs(w.i));
        n++;

        //      cout << "XY = " << xy << endl;
        //      cout << "XX = " << xx << endl;
        //       cout << "REal t = " << t.r << endl;
        //      cout << "Iter NN, t = ";
        //       Print(&t);
   }

   if (n == 11)
      cout << "\a\a\a Warninng: iteration limit reached, result of W may be inaccurate" << endl;

/*    cout << " n = " << n << endl;
      cout << "\n\n***** CACULATED VALUE of LambertW function: "; Print(&w);
      cout << endl;
      cout << "LambertW function returns value: "
*/

  return w.r;
} // end of LambertW routine

// end of Cbsim.cpp
```

Appendix C

Data Manipulation Algorithms

GridEdge.aml

```
/* gridedge.aml
/*
/* An insanely simple algorithm which corrects one pixel gaps
/* found when connecting USGS DEMs. Any irregular gaps (i.e. diagonals
/* should be corrected first by using Arcedit.
/*
/* smr - 10/12/99

DOCELL
if (isnull(barclip)) output = (barclip(-1,0) + barclip(1,0)) DIV 2
        else output = barclip
END

/* end of gridedge.aml
```

Xmrgtoascster.c

```
/*

Name: xmrgtoascster.c


Description: Read an XMRG file and write to an ASCII file that

can be read directly read by ArcView as an Arc/Info grid. Output

coordinates are polar stereographic.


This program will recognize two types of XMRG headers

-- pre and post AWIPS Bld 4.2

Also modified in Jan. 2000 to recognize pre-1997 headers which don't

have a second record in the header.

Successfully compiled and run on NHDRS using the following syntax:

cc -g -Aa -o xmrgtoascster xmrgtoascster.c

Syntax to run the program is then:

xmrgtoascster <infilename> <outfilename>

-- Do not include an extension in the output file name.
-- Developed by Seann Reed, NWS

*/


#include <stdio.h>
#include <math.h>
#include <stdlib.h>

main(int argc,char *argv[])

{

        FILE     *in_file_ptr, *out_file_ptr;

        char     binfile[ 18] , outfile[ 18] ;
```

```
    char     tempstr[ 256] , user_id[ 10] , date[ 10] , time[ 10] , process_flag[ 8] ;
    char date2[ 10] ,time2[ 10] ;
    char dummy[ 10] ,asc_name[ 18] ;

    int     rfchd[ 4] ;
    int     ddd[ 2] ;
    int numsuccess,*numbytes;

short *itest;

    long     MAXX, MAXY, XOR, YOR;
    long     nrows, ncols;
    long     i, j, temp;
    /*short     precip[ 1000] ;*/
    short  *onerow;
    /*int     rainfall[ 1000][ 1000] ;*/
    float **matrix;
    float outval;
    float xstereo,ystereo;


    /* end variable declaration */

if (argc != 4)
    {
       (void)printf("Incorrect number of arguments. Should be 4.\n");

   exit(0);

}

in_file_ptr=fopen(argv[ 1] ,"rb");

if (in_file_ptr == NULL)

   {

   (void)printf("Can not open file %s for input.\n",argv[ 1] );

            return(1);

   }

(void)strcpy(asc_name,argv[ 2] );

(void)strcat(asc_name,".asc");

out_file_ptr=fopen(asc_name,"w");

if (out_file_ptr == NULL)

   {

   (void)printf("Can not open file %s for output.\n",argv[ 2] );

   return(1);

   }


/* start reading the XMRG file*/

    /*SEEK_SET specifies the position offset from the beginning of the file*/

    fseek(in_file_ptr, 4, SEEK_SET);

    for(i=0;i<4;i++)

    {
```

```
            fread(&rfchd[ i], sizeof(int), 1, in_file_ptr);
    }

    XOR=rfchd[ 0] ;
    YOR=rfchd[ 1] ;
    xstereo=XOR*4762.5-401.0*4762.5;
    ystereo=YOR*4762.5-1601.0*4762.5;
    MAXX=rfchd[ 2] ;
    MAXY=rfchd[ 3] ;
    nrows = MAXY;
    ncols = MAXX;

    /*print to header file*/
    (void)fprintf(out_file_ptr,"ncols %d\n",MAXX);
    (void)fprintf(out_file_ptr,"nrows %d\n",MAXY);
    /*echo to screen*/
    (void)printf("ncols %d\n",MAXX);
    (void)printf("nrows %d\n",MAXY);
    if (strcmp(argv[ 3] ,"hrap")==0)
    {
        (void)fprintf(out_file_ptr,"xllcorner %d\n",XOR);
        (void)fprintf(out_file_ptr,"yllcorner %d\n",YOR);
        (void)fprintf(out_file_ptr,"cellsize 1\n");
            (void)printf("xllcorner %d\n",XOR);
        (void)printf("yllcorner %d\n",YOR);
        (void)printf("cellsize 1\n");
    }
else if (strcmp(argv[ 3] ,"ster")==0)
    {
        (void)fprintf(out_file_ptr,"xllcorner %f\n",xstereo);
        (void)fprintf(out_file_ptr,"yllcorner %f\n",ystereo);
        (void)fprintf(out_file_ptr,"cellsize 4762.5\n");
        (void)fprintf(out_file_ptr,"nodata_value -9999.0\n");
        (void)printf("xllcorner %f\n",xstereo);
        (void)printf("yllcorner %f\n",ystereo);
        (void)printf("cellsize 4762.5\n");
```

```
        /*nodata_value and byteorder are optional*/
        /*echo to screen*/
    }
      else
      {
         (void)printf("Specify either hrap or ster as the third argument.\n");
      }

  /*each record is preceded and followed by 4 bytes*/
      /*first record is 4+16+4 bytes*/
      fseek(in_file_ptr, 24, SEEK_SET);
      /*read second FORTRAN record*/
      fread(&numbytes,4,1,in_file_ptr);
      fseek(in_file_ptr, 4, SEEK_CUR);

      numsuccess=fscanf(in_file_ptr, "%10s %10s %10s %8s %10s %10s", user_id, date, time,
process_flag,date2,time2);
      /*numsuccess=fscanf*/

      /*first record (24) plus second record(46) is 70*/
      /*if (strlen(date2)>0)*/
      if ((int) numbytes == 66)
      {
            fseek(in_file_ptr, 98, SEEK_SET);
            (void)printf("user_id %10s\n",user_id);
         (void)printf("date %10s\n",date);
         (void)printf("time %10s\n",time);
         (void)printf("process_flag %8s\n",process_flag);
         (void)printf("datelen %d\n",strlen(date));
         (void)printf("timelen %d\n",strlen(time));
         (void)printf("user_id %d\n",strlen(user_id));
         (void)printf("date2 %s\n",date2);
         (void)printf("time2 %s\n",time2);
         (void)printf("numbytes %d\n",numbytes);
      }
```

```
    else if ((int) numbytes==38)
    {
            fseek(in_file_ptr, 70, SEEK_SET);
            /* (void)printf("gothere\n");*/
            (void)printf("user_id %10s\n",user_id);
       (void)printf("date %10s\n",date);
       (void)printf("time %10s\n",time);
       (void)printf("process_flag %8s\n",process_flag);
       (void)printf("numbytes %d\n",numbytes);
}
    else
    {
            (void)printf("Reading pre-1997 format.\n");
            fseek(in_file_ptr,24, SEEK_SET);
    }

    /* allocate memory for arrays */
    onerow = (short int*) malloc(sizeof(short int*)*ncols);
    matrix = (float**) malloc(sizeof(float*)*nrows);
for (i=0;i<nrows;i++)
   matrix[ i] =(float*) malloc(sizeof(float)*ncols);

    for(i=nrows-1;i>-1;i--)
    {
    fseek(in_file_ptr, 4, SEEK_CUR);
    /* read one row */
    fread(onerow,sizeof(short),ncols,in_file_ptr);
    fseek(in_file_ptr, 4, SEEK_CUR);
            for(j=0;j<ncols;j++)
            {
            matrix[ i][ j] = (float) onerow[ j];

            } /* close j */
    } /* close i  */

    for(i=0; i<nrows; i++)
```

```
        {
                for(j=0; j<ncols; j++)
                {
                        /*fwrite(&rainfall[ i][ j], 4, 1, out_file_ptr);*/
                        outval=matrix[ i][ j];
                        if (matrix[ i][ j] < 0)
                        {
                                outval=-9999.0;
                        }
                        else
                        {
                           outval = outval/1000.0;
                                /*  convert from hundredths of mm to cm*/
                    }
          /*fwrite(&outval,4,1,out_file_ptr);*/
                        fprintf(out_file_ptr,"%f ",outval);
                        }
                        fprintf(out_file_ptr,"\n");
        }

/*free allocated memory*/
free(onerow);
for (i=0;i<nrows;i++)
                { free(matrix[ i]);free(matrix); }
fclose(in_file_ptr);
fclose(out_file_ptr);
/*fclose(hdr_file_ptr);*/


}  /**  END OF MAIN  **/
```

Radarconv.aml

```
/* radarconv.aml
/* takes the raw XMRG tar files and runs through a series of steps
/* 1) untar and unzip the radar files
/* 2) run xmrgtoascster to convert the binary files
/* 3) move the data to ARC/INFO grid format
/* 4) Check for errors in each grid
/* 5) Project grid to UTM coordinate system
```

```
/* Important Notes: This algorithm will have problems in the following cases:
/*     - When the tar names change! Make changes as needed
/*     - When the data does not exist!

/* Developed by smr, 11/2/99
/* Last changed - 2/15/00

/* must be used in the following manner...
/* &r radarconv year rfc
/*

&args year rfc
&sv holder = 0

&do cmonth = 1 &to 12 &by 1

&if %cmonth% < 10 &then
        &sv month = 0%cmonth%
&else
        &sv month = %cmonth%

&sys tar -xvf /d4/scottradar/rawdata/Siii%month%19%year%%rfc%.tar

&if %cmonth% = 4 OR %cmonth% = 6 OR %cmonth% = 9 OR %cmonth% = 11 &then
        &sv cday = 30
&else
        &if %cmonth% = 2 &then
                &if %year% = 96 &then
                        &sv cday = 29
                &else
                        &sv cday = 28
        &else
                &sv cday = 31

&do date = 1 &to %cday% &by 1

&if %date% < 10 &then
        &sv day = 0%date%
&else
        &sv day = %date%

/* for 1995, there is another problem, add the following line...

/*&sys mv rawdata/Siii%month%%day%%year%%rfc%.tar /d4/scottradar

/* end of modification...

&sys tar -xvf /d4/scottradar/Siii%month%%day%19%year%%rfc%.tar

&do time = 0 &to 23 &by 1

&if %time% < 10 &then
        &sv hr = 0%time%
&else
        &sv hr = %time%

/* This is truly strange-The Beginning of October of 1999 uses a
/* different tar structutre which puts results in /d4 not /d4/scottradar
/* for those, add the line which follows this: (ends @ 10/27/99)

/* &sys mv /d4/xmrg%month%%day%%year%%hr%z.Z xmrg%month%%day%%year%%hr%z.Z

/* end of change for weird part of month...

/* Another change-for 10/28/99 on need to add the 19 in front of year...

&sys gunzip /d4/scottradar/xmrg%month%%day%%year%%hr%z.Z
&sys xmrgtoascster xmrg%month%%day%%year%%hr%z peach ster

asciigrid peach.asc peagrid float
```

```
&describe peagrid

&if %grd$xmin% > 0 &then &do
&type xmrg%month%%day%%year%%hr% is screwed up...
kill peagrid
&end

&else &do
projectdefine grid peagrid
projection polar
spheroid sphere
units meters
parameters
-105 0 0
60 0 24.5304792
0
0
project grid peagrid pearadar
output
projection utm
zone 15
spheroid clarke1866
parameters
end

grid
gridclip pearadar pclip box 344634 3980000 ~
       360000 3996000
quit

gridascii pclip /d4/scottradar/ascii/19%year%/p%month%%day%%year%%hr%.txt

kill peagrid
kill pearadar
kill pclip
&sys rm peach.asc
&sys rm -f xmrg%month%%day%%year%%hr%z
/* &sys rm /d4/scottradar/ascii/19%year%/p%month%%day%%year%%hr%.prj

&end

&end
&end
&end

&return
```

Commands.aml

```
/* commands.aml

/* Commands for creating the voronoi polygons from tRIBS output
/* The input name can change, i.e. replace toutput_voi with any
/*    "_voi" file from tRIBS.

&wo /d4/scottarc
generate trivoi
input toutput_voi
polygons
quit
clean trivoi tribase # # POLY
build tribase POLY
```

Rainconvert.aml

```
/* rainconvert.aml

/* Converts the gridded rainfall to fit the voronoi polygons of the
/* RIBS model using erain.aml

/* Caution: Do not use this for other datasets. Conversion factors
/* for rainfall are built in!

/* must be used in the following manner...
/* &r rainconvert voicover smth sday emth eday

&args voicover smth sday emth eday

&sv year = 96

&do cmonth = %smth% &to %emth% &by 1

&if %cmonth% < 10 &then
        &sv month = 0%cmonth%
&else
        &sv month = %cmonth%

&do cdays = %sday% &to %eday% &by 1

&if %cdays% < 10 &then
        &sv days = 0%cdays%
&else
        &sv days = %cdays%

&do chours = 0 &to 23 &by 1

&if %chours% < 10 &then
        &sv hours = 0%chours%
&else
        &sv hours = %chours%

cp /d4/scottradar/ascii/19%year%/%month%/p%month%%days%%year%%hours%.txt ~
/d4/scottarc/

asciigrid p%month%%days%%year%%hours%.txt p%month%%days%%year%%hours%g float

grid
p%month%%days%%year%%hours%g2 = p%month%%days%%year%%hours%g * 1000
p%month%%days%%year%%hours%g3 = int(p%month%%days%%year%%hours%g2)
p%month%%days%%year%%hours%c = gridpoly(p%month%%days%%year%%hours%g3)
quit

&r erain %voicover% p%month%%days%%year%%hours%c

copy voiadded rain/p%month%%days%%year%%hours%c2

rm p%month%%days%%year%%hours%.txt
kill p%month%%days%%year%%hours%g
kill p%month%%days%%year%%hours%g2
kill p%month%%days%%year%%hours%g3
kill p%month%%days%%year%%hours%c
kill voiadded
kill identest

&end

&end

&end

&return
```

Erain.aml

```
/* erain.aml

/* Converts rainfall to the voronoi cells of RIBS

/* must be used in the following manner:
/* &r erain voicover raincover

&args voicover raincover
copy %voicover% voiadded

ae  /* enter arcedit to add the item effrain to voiadded
edit voiadded
ef polys
additem effrain 4 12 F 3
quit;Y;Y /* exiting arcedit

identity voiadded %raincover% identest POLY

&describe voiadded
&sv numpoly = %DSC$POLYGONS% /* this tells us what to loop through

cursor voiarea declare voiadded.pat INFO RW
cursor voiarea open

&sv start = 2
&do counter = %start% &to %numpoly% &by 1
        &sv cumrain = 0
cursor voiarea next
cursor polysliv declare identest polys RW VOIADDED# = %counter%
cursor polysliv open
&do &while %:polysliv.aml$next%
        &sv cumrain = [ calc %cumrain% + %:polysliv.area% / %:voiarea.area% ~
* %:polysliv.grid-code%]
cursor polysliv next
&end
cursor polysliv remove
&sv :voiarea.effrain = %cumrain% / 1000
&end

cursor voiarea remove

&return
```

Egwater.aml

```
/* egwater.aml
/* must be used in the following manner:
/* &r egwater voicover gwtcover

/* Very similar to erain-with one big difference-
/* this program just picks one of the grid values and uses it
/* this is due to problems with NO DATA values (i.e. -9999)

&args voicover raincover
copy %voicover% voiadded

ae  /* enter arcedit to add the item effrain to voiadded
edit voiadded
ef polys
additem effrain 4 12 F 3
quit;Y;Y /* exiting arcedit

identity voiadded %raincover% identest POLY

&describe voiadded
```

```
&sv numpoly = %DSC$POLYGONS% /* this tells us what to loop through

cursor voiarea declare voiadded.pat INFO RW
cursor voiarea open

&sv start = 2
&do counter = %start% &to %numpoly% &by 1
        &sv cumrain = 0
cursor voiarea next
cursor polysliv declare identest polys RW VOIADDED# = %counter%
cursor polysliv open
&do &while %:polysliv.aml$next%
  &if %:polysliv.grid-code% > -1 &then &do
          &sv cumrain = %:polysliv.grid-code%
    cursor polysliv next
  &end
  &else
    cursor polysliv next
&end
cursor polysliv remove
&sv :voiarea.effrain = %cumrain%
&end

cursor voiarea remove

&return
```

Tribscov.aml

```
/* tribscov.aml

/* Used to move tRIBS output to ARC/INFO datasets.

/* must be used in the following manner:
/*  tribscov basename timehr timemin

&args basename timehr timemin
&wo /d4/scottarc
&sv viewname = B%timehr%_%timemin%
&sv fullname = %basename%_%timehr%_%timemin%
copy %basename% %viewname%
&data arc info
ARC
DEFINE STUFF.DAT
%viewname%-ID
4
5
B
ZVAL
8
9
F
3
NWTVAL
8
9
F
3
SRF
8
9
F
3
NFVAL
8
9
F
3
```

```
NTVAL
8
9
F
3
~
ADD FROM /D4/SCOTTARC/%fullname%
Q STOP
&end
joinitem %viewname%.pat stuff.dat %viewname%.pat %viewname%-id
&data arc info
ARC
SELECT STUFF.DAT
DELETE STUFF.DAT
Y
Q STOP
&end
```

Createribs.cpp

```
// createribs.cpp
// This is a fairly simple C++ program designed to take the arcview files
// produced and make them into one file ready for ribs...
// In the directory where this program resides, place the following…
// table5.txt = interior VIP points (2% selection)
// table4.txt = interior VIP points (1% selection)
// table3.txt = interior 'ring' points
// table2.txt = boundary points
// table1.txt = outlet point
// table7.txt = stream points(densify @ 180m)
//
// the file that comes out is test.points
// Note: look at the screen output to get number of pts, and close points!
//       these pts will be doubled. (12 listed = 6 bad points)

#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <string.h>
#include <iomanip.h>
#include <math.h>

main()
{
  ifstream infile ("table4.txt");
  ofstream outfile ("test.points");
  ofstream outfile2 ("test2.points");

  if (!infile) {
    cerr << "You screwed up, file doesn't exist" << endl;
    exit(1);
  }

       char oneline[ 90] ;
  char *tokenPtr;

  infile >> oneline; // junks the header

  while (infile >> oneline){

         tokenPtr = strtok( oneline,",");

         while (tokenPtr != NULL){
                outfile << setprecision(12) << tokenPtr << " ";
                     outfile2 << setprecision(12) << tokenPtr << " ";
     tokenPtr = strtok( NULL, ",");
    }
              outfile << " 0" << endl;
```

```
  outfile2 << " 0" << endl;
}

infile.close();

      infile.open("table3.txt");

infile >> oneline; // junks the header

while (infile >> oneline){

        tokenPtr = strtok( oneline,",");

        while (tokenPtr != NULL){
                outfile << setprecision(12) << tokenPtr << " ";
    outfile2 << setprecision(12) << tokenPtr << " ";
    tokenPtr = strtok( NULL, ",");
  }
             outfile << " 0" << endl;
  outfile2 << " 0" << endl;
}

infile.close();

      infile.open("table7.txt");

infile >> oneline; // junks the header

while (infile >> oneline){

        tokenPtr = strtok( oneline,",");

        while (tokenPtr != NULL){
                outfile << setprecision(12) << tokenPtr << " ";
    outfile2 << setprecision(12) << tokenPtr << " ";
    tokenPtr = strtok( NULL, ",");
  }
             outfile << " 3" << endl;
  outfile2 << " 3" << endl;
}

      infile.close();

      infile.open("table2.txt");

infile >> oneline; // junks the header

while (infile >> oneline){

        tokenPtr = strtok( oneline,",");

        while (tokenPtr != NULL){
                outfile << setprecision(12) << tokenPtr << " ";
    outfile2 << setprecision(12) << tokenPtr << " ";
    tokenPtr = strtok( NULL, ",");
  }
             outfile << " 1" << endl;
  outfile2 << " 1" << endl;
}

infile.close();

      infile.open("table1.txt");

infile >> oneline; // junks the header

while (infile >> oneline){

        tokenPtr = strtok( oneline,",");

        while (tokenPtr != NULL){
```

```
                outfile << setprecision(12) << tokenPtr << " ";
      outfile2 << setprecision(12) << tokenPtr << " ";
      tokenPtr = strtok( NULL, ",");
    }
              outfile << " 2" << endl;
    outfile2 << " 2" << endl;
  }

  infile.close();
       outfile.close();
  outfile2.close();

  ifstream infile2;
       infile.open("test.points");

       int numpts, dblcount, bound, bound2;
       double x, y, z, x2, y2, z2;
  numpts = 0;

       while(infile >> x >> y >> z >> bound){
    numpts++;
    dblcount = 0;
              infile2.open("test2.points");
              while(infile2 >> x2 >> y2 >> z2 >> bound2){
                      if ((fabs(x2-x)<1) && (fabs(y2-y)<1))
       dblcount++;
    }
    if (dblcount > 1)
      cout << "remove " << x << ',' << y << ',' << z << ',' << bound << endl;
    infile2.close();
  }
  cout << "The number of pts is ..." << numpts << endl;
}
```

Bibliography

[1] M.B. Abbott, J.C. Bathurst, J.A. Cunge, et al. An introduction to the european hydrological system-systeme hydrologique europeen,"SHE",1:history and philosophy of a physically based, distributed modeling system. *Journal of Hydrology*, 87:45–59, 1986a.

[2] M.B. Abbott, J.C. Bathurst, J.A. Cunge, et al. An introduction to the european hydrological system-systeme hydrologique europeen,"SHE",2:structure of a physically based, distributed modeling system. *Journal of Hydrology*, 87:61–77, 1986a.

[3] J. Amorocho and G.T. Orlob. *Nonlinear Analysis of Hydrologic Systems.* University of California Water Resources Center, Contribution No. 40, November 1961.

[4] M.G. Anderson and T.P. Burt. Modelling strategies. In M.G. Anderson and T.P. Burt, editors, *Hydrological Forecasting*, pages 1–13. John Wiley & Sons, Inc., Chichester, England, 1985.

[5] J.C. Bathurst and P.E. O'Connell. Future of distributed modelling: The systeme hydrologique europeen. In K.J.Beven and I.D.Moore, editors, *Terrain Analysis and Distributed Modeling in Hydrology*, chapter 13, pages 213–226. John Wiley & Sons, Inc., Chichester, England, 1993.

[6] Keith Beven. Infiltration into a class of vertically non-uniform soils. *Journal of Hydrologic Science*, 29(4):425–434, 1984.

[7] Keith J. Beven and Andrew Binley. The future of distributed models:model calibration and uncertainty prediction. In K.J.Beven and I.D.Moore, editors, *Terrain Analysis and Distributed Modeling in Hydrology*, chapter 14, pages 227–246. John Wiley & Sons, Inc., Chichester, England, 1993.

[8] Keith J. Beven and Mike J. Kirkby. A physically-based, variable contributing area model of basin hydrology. *Hydrologic Sciences Bulletin*, 24(1):43–69, 1979.

[9] J. Braun and M. Sambridge. Modeling landscape evolution on geological time scales: a new mehtod based on irregular spatial discretization. *Basin Research*, 9:27–52, 1997.

[10] R.H. Brooks and A.T. Corey. *Hydraulic properties of porous media, Hydrology Paper 3.* University of Colorado, Fort Collins, Colorado, 1964.

[11] J.C. Burnash, R.L. Ferral, and R.A. McGuire. *A generalized Streamflow Simulation System, Conceptual Modeling for Digital Computers.* Joint Federal-State Forecast Center, NOAA National Weather Service, and State of California Dept. of Water Resources, March 1973.

[12] Mariza C. Cabral, L. Garrote, R.L. Bras, and D. Entekhabi. A kinematic model of infiltration and runoff generation in layered and sloped soils. *Advances in Water Resources*, 15:311–324, 1992.

[13] National Soil Survey Center. *State Soil Geographic (STATSGO) Data Base: Data Use Information.* US Department of Agriculture, July 1994.

[14] R.B. Clapp and G.M. Hornberger. Empirical equations for some soil hydraulic properties. *Water Resources Research*, 14(4):601–604, 1978.

[15] Mariza C. Costa-Cabral and Stephen J. Burges. Digital elevation model networks (DEMON): A model of flow over hillslopes for computation of contributing and dispersal areas. *Water Resources Research*, 30:1681–1692, June 1994.

[16] National Research Council. *Towards a New National Weather Service:Assessment of Hydrologic and Hydrometerological Operations and Services.* National Weather Service Modernization Committee, 1996.

[17] W.E.H. Culling. Analytical theory of erosion. *Journal of Geology*, 68:336–344, 1960.

[18] G. Dagan and E. Bressler. Unsaturated flow in spatially variable fields. 1-derivation of models of infiltration and redistribution. *Water Resources Research*, 19(2):413–420, 1983.

[19] J.C.I. Dooge. A general theory of the unit hydrograph. *Journal of Geophysical Research*, 64(1):241, 1957.

[20] T. Dunne and R.D. Black. Partial area contributions to storm runoff in a small new england watershed. *Water Resources Research*, 6(5):1296–1311, 1970.

[21] T. Dunne, T.R. Moore, and C.H. Taylor. Recognition and prediction of runoff producing zones in humid regions. *Hydrologic Sciences Bulletin*, 20:305–327, 1975.

[22] EPA. *EPA Reach File Version 3.0 Alpha Release (RF3-Alpha) Technical Reference.* Washington, DC, December 1994.

[23] H.A. Foster. Theoretical frequency curves and their application to engineering problems. *Transactions, ASCE*, 87:142–173, 1924.

[24] R.A. Freeze and R.L. Harlan. Blueprint for a physically-based digitally-simulated hydrologic response model. *Journal of Hydrology*, 9:237–258, 1969.

[25] W. Gandoy-Bernasconi and O.L. Palacios-Velez. Automatic cascade numbering of unit elements in distributed hydrologic models. *Journal of Hydrology*, 112:375–393, 1990.

[26] L. Garrote and R.L. Bras. A distributed model for real-time flood forecasting using digital elevation models. *Journal of Hydrology*, 167:279–306, 1995.

[27] Luis Garrote. Real-time modeling of river basin response using radar-generated rainfall maps and a distributed hydrologic database. Masters and engineers thesis, Massachusetts Institute of Technology, January 1993.

[28] W.J. Gbureck. Initial contributing area of a small watershed. *Journal of Hydrology*, 118:387–403, 1990.

[29] Craig Goodwin and David Tarboton. *SDTS DEM to ArcView Grid Conversion Utility.* Utah State University, Logan, Utah, 2000.

[30] R.B. Grayson, I.D. Moore, and T.A. McMohan. Physically based hydrologic modeling, 2, is the concept realistic? *Water Resources Research*, 28(10):2659–2666, October 1992.

[31] W.H. Green and G.A. Ampt. Studies of soil physics, 1, flow of air and water through soils. *Journal of Agricultural Science*, 4:1–24, 1911.

[32] L. Guibas and J. Stolfi. Primitives for the manipulations of general subdivisions and the computation of voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, 1985.

[33] V. Gupta. A representation of an instantaneous unit hydrograph from geomorphology. *Water Resources Research*, 15(6):855–862, 1980.

[34] A. Hazen. Discussion on 'flood flows' by w.e. fuller. *Transactions, ASCE*, 77:628, 1914.

[35] R.E. Horton. Frequency of recurrence of hudson river floods. *US Weather Bureau Bulletin*, Z:109–112, 1913.

[36] R.E. Horton. *Surface Runoff Phenomena. Part 1-Analysis of the Hydrograph.* Horton Hydrological Laboratory Publication 101, Voorhessville, NY, 1935.

[37] R.E. Horton. Erosional development of streams and their drainage basins: hydrophysical approach to quantitative geomorphology. *Bull. Geological Society Am.*, 56:275–370, 1945.

[38] Federal Systems Integration and Management Center. *The Spatial Data Transfer Standard: Guide for Technical Managers.* SDTS Task Force, 1996.

[39] V.Y. Ivanov. Personal communication. This will be turned into a thesis by Dec 2000., June 2000.

[40] T.H. Jackson, D.G. Tarboton, and K.R. Cooley. A spatially-distributed hydrologic model for a small arid mountain watershed. *Unpublished*, May 1996.

[41] N.L. Jones, S.G. Wright, and D.R. Maidment. Watershed delineation with triangle-based models. *Journal of Hydraulic Engineering*, 116:1232–1251, 1990.

[42] P.Y. Julien, B. Saghafian, and Fred L. Ogden. Raster-based hydrologic modeling of spatially-varied surface runoff. *Water Resources Bulletin*, 31(3):523–536, 1995.

[43] D.E. Knuth. *Axioms and Hulls, Lecture notes in computer science, no. 606.* Springer-Verlag, New York, 1992.

[44] R.K. Linsley and N.H. Crawford. Computation of a synthetic streamflow record on a computer. *Hydrol. Science Bulletin, IASH*, 51:526–538, 1960.

[45] D. Scott Mackay and Lawrence E. Band. Extraction and representation of nested catchment areas from digital elevation models in lake-dominated topography. *Water Resources Research*, 34:897–901, April 1998.

[46] David M. Mark. Network models in geomorphology. In *Modelling in Geomorphological Systems.* John Wiley & Sons, Inc., 1988.

[47] Metcalf and Eddy. *Storm water management model.* US EPA Report 110224DOC, 1971.

[48] Y. Mualem. Hydraulic conductivity of unsaturated porous media: Generalized macroscopic approach. *Water Resources Research*, 14(2):325–334, 1978.

[49] T.J. Mulvaney. On the use of self-registering rain and flood gauges in making observations of the relations of rainfall and of flood discharges in a given catchment. *Transactions of the Institution of Civil Engineers, Ireland*, 4(2):18, 1851.

[50] J.E. Nash. The form of instantaneous unit hydrograph. *Hydrol. Science Bulletin*, 3:114–121, 1959.

[51] E.J. Nelson, N.L. Jones, and A.W. Miller. Algorithm for precise drainage basin delineation. *Journal of Hydraulic Engineering*, 120:298–312, 1994.

[52] S.P. Neuman. Wetting front pressure head in the infiltration model of green and ampt. *Water Resources Research*, 12(3), 1976.

[53] John F. O'Callaghan and David M. Mark. The extraction of drainage networks from digital elevation data. *Computer Vision, Graphics, and Image Processing*, 28:323–344, 1984.

[54] US Army Corps of Engineers. *HEC-1, Flood Hydrograph Package.* Hydrologic Engineering Center, Davis, California, October 1968.

[55] Fred L. Ogden. *CASC2D Reference Manual.* Univ. of Connecticut, Storrs, Connecticut, 1998.

[56] Fred L. Ogden and S.U.S. Senarath. Continuous distributed-parameter hydrologic modeling with casc2d. In *Proceedings of XXVII IAHR Congress*, pages 864–869. Int. Assoc. for Hydraul. Res., Delft, Netherlands, 1997.

[57] O.L. Palacios-Velez and B. Cuevas-Renaud. Automated river-course, ridge and basin delineation from digital elevation data. *Journal of Hydrology*, 86:299–314, 1986.

[58] Seann M. Reed. Displaying and using NWS XMRG/HRAP files within ArcView or Arc/Info GIS. Internal National Weather Service-Hydrologic Research Center research paper, January 2000.

[59] Seann M. Reed and David R. Maidment. Coordinate transformations for using NEXRAD data in GIS-based hydrologic modeling. *Journal of Hydrologic Engineering*, 4(2):174–182, April 1999.

[60] Jens Christian Refsgaard. Model and data requirements for simulation of runoff and land surface processes in relation to global circulation models. In *Global Environmental Change and Land Surface Processes in Hydrology: The Trials and Tribulations of Modeling and Measuring.* Springer-Verlag, New York, NY, 1994.

[61] I. Rodriquez-Iturbe and J. Valdez. The geomorphologic structure of hydrologic response. *Water Resources Research*, 15(6):1409–1420, 1979.

[62] M. Sambridge, J. Braun, and H. McQueen. Geophysical parameterization and interpolation of irregular data using natural neighbors. *Geophysical Journal International*, 122:837–857, 1995.

[63] Sharika U.S. Senarath et al. On the calibration and verification of two-dimensional, distributed, hortonian, continuous watershed models. *Water Resources Research*, 36(6):1495–1510, June 2000.

[64] L.K. Sherman. Streamflow from rainfall by unit-graph method. *Engineering News Record*, 108:501–505, 1932.

[65] J. Simpson et al. Eyeing the eye: Exciting early stage science results from TRMM. *Bulletin of the American Meterological Society*, 79(8), August 1998.

[66] M. Sivapalan, K. Beven, and E. Wood. On hydrologic similarity: 2. a scaled model of storm runoff production. *Water Resources Research*, 23(12):2266–2278, December 1987.

[67] James A. Smith, D.J. Seo, M.L. Baeck, and M.D. Hudlow. An intercomparison study of NEXRAD precipitation estimates. *Water Resources Research*, 32(7):2035–2045, July 1996.

[68] Michael B. Smith, Victor Koren, et al. *Distributed Modeling: Phase 1 Results.* NOAA Technical Report NWS 44. U.S. Department of Commerce, February 1999.

[69] R.E. Smith, C. Corradini, and F. Melone. Modeling infiltration for multistorm runoff events. *Water Resources Research*, 29(1):133–144, January 1993.

[70] U.S. Geological Survey. *Standards for Digital Elevation Models*, chapter 2: Specifications. U.S. Department of the Interior, 1998.

[71] S.W.Sloan. A fast algorithm for constructing delaunay triangulations in the plane. *Advances in Engineering Software*, 9(1):34–55, 1987.

[72] David G. Tarboton. A new method for the determination of flow directions and upslope areas in grid digital elevation models. *Water Resources Research*, 33:309–319, February 1997.

[73] David G. Tarboton, Rafael L. Bras, and Ignacio Rodriguez-Iturbe. On the extraction of channel networks from digital elevation data. In K.J.Beven and I.D.Moore, editors, *Terrain Analysis and Distributed Modeling in Hydrology*, chapter 5, pages 85–104. John Wiley & Sons, Inc., Chichester, England, 1993.

[74] G.E. Tucker, S.T. Lancaster, N.M. Gasparini, R.L. Bras, and S.M. Rybarczyk. An object-oriented framework for distributed hydrologic and geomorphic modeling using triangulated irregular networks. *Computers and Geosciences*, in press, 2000.

[75] Gregory E. Tucker, S.T. Lancaster, N.M. Gasparini, and R.L. Bras. The channel-Hillslope Integrated Landscape Development (CHILD) model. In R.S. Harmon and W.W. Doe, editors, *Landscape Erosion and Sedimentation Modeling.* Kluwer Press, Submitted May, 2000.

[76] Hydrology Subcommittee US Interagency Advisory Committee on Water Data. *Guidelines for Determining Flood Flow Frequency.* Bulletin No. 17B, Reston, Virginia, 1983.

[77] D.F. Watson and G.M. Philip. Systematic triangulations. *Computer Vision, Graphics, and Image Processing*, 26:217–223, 1984.

[78] J.M. Wicks and J.C. Bathurst. SHESED:a physically based, distributed erosion and sediment yield component for the she hydrological modelling system. *Journal of Hydrology*, 175(1-4):213–238, 1996.

[79] T.C.J. Yeh, L.W. Gelhar, and A.J. Gutjahr. Stochastic analysis of unsaturated flow in heterogenous soils. 1-statistically isotropic media. *Water Resources Research*, 21(4):447–456, 1985.

[80] C. Bryan Young et al. An evaluation of NEXRAD precipitation estimates in complex terrain. *Journal of Geophysical Research*, 104(D16):19691–19703, August 1999.