INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®

A NUMERICAL MODEL AND SEMI-ANALYTIC EQUATIONS FOR DETERMINING WATER TABLE ELEVATIONS AND DISCHARGES IN NON-HOMOGENEOUS SUBSURFACE DRAINAGE SYSTEMS

By

Armando Uribe-Chavez

A Dissertation Submitted to the Faculty of the

DEPARTMENT OF AGRICULTURAL AND BIOSYSTEMS ENGINEERING

In Partial Fulfillment of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY

In the Graduate College

THE UNIVERSITY OF ARIZONA

2001

UMI Number: 3010252

UMI Microform 3010252

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

THE UNIVERSITY OF ARIZONA ®
GRADUATE COLLEGE

As members of the Final Examination Committee, we certify that we have read the dissertation prepared by Armando Uribe-Chavez

entitled A numerical model and semi-analytic equations for determining water table elevations and discharges in non-homogeneous subsurface drainage.

and recommend that it be accepted as fulfilling the dissertation requirement for the Degree of Doctor of Philosophy

Peter Waller	02/02/01 Date
Donald Slack	02/02/01 Date
Muluneh Yitayew	02/02/01 Date
Arthur W. Warrick	02/02/01 Date
Jim Yen	02/02/01 Date

Final approval and acceptance of this dissertation is contingent upon the candidate's submission of the final copy of the dissertation to the Graduate College.

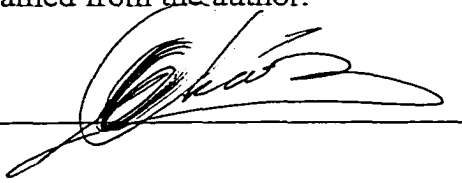
I hereby certify that I have read this dissertation prepared under my direction and recommend that it be accepted as fulfilling the dissertation requirement.

Dissertation Director Peter Waller — 02/02/01 Date

STATEMENT by AUTHOR

This dissertation has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this dissertation are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his or her judgment the proposed use of the material is in the interest of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: 

ACKNOWLEDGMENTS

I specially want to thank Dr. Peter Waller for his advice and help to make possible this dissertation.

A deep recognition to the Department of Agricultural and Biosystems Engineering where I found constant encouragement, in particular to its head and member of my committee, Dr. Don Slack.

I should like to express my gratitude to Dr. Arthur Warrick and Dr. Muluneh Yitayew for their comments to improve the draft.

My gratitude also goes to Dr. Jim Yeh for serving as committee member.

I give thanks to Dr. Waldo Ojeda for his suggestions and assistance in reviewing the manuscript.

Thanks to M.C. Heriberto Estrella for his help given to me to begin the doctorate programme.

Also I want to say thank to Dr. Sylvia Ortega for her aid provided to me to continue my studies.

I am grateful with Dr. Adela Allen and Mrs. Geraldine Olds for their great help offered to me to finish my doctorate.

Finally, I appreciate the scholarships given to me by the Mexican agency CONACYT and the association ANUIES, and the support granted to me by the Autonomous University of Chapingo.

DEDICATION

To my parents Luis and Maria Trinidad

To my children Minerva and Ulises.

To my brothers and sisters: Luis, Olga, Lilia, Rafael, Arturo, Alfredo, and Elsa.

To Nora

To the people of Mexico who sustained my education.

TABLE OF CONTENTS

Page

LIST OF FIGURES 9

LIST OF TABLES 12

ABSTRACT 13

1. INTRODUCTION 15

2. LITERATURE REVIEW 19
   2.1. Analytical Solutions 19

   2.2. Physical Models and Experiments 22

   2.3. Numerical Models 23

3. THEORETICAL BACKGROUND 28
   3.1. Finite Element Method 28
      3.1.1. Governing Differential Equation 28
      3.1.2. Galerkin's Method 29
      3.1.3. Boundary Conditions 34
      3.1.4. Adaptive Mesh 37
      3.1.5. Water Table Position Differential Equation and Drainable Porosity Equation 37
      3.1.6. Entry Losses 42

   3.2. Drain System Equations to Predict Water Table Fall and Discharge 42

4. THEORETICAL DEVELOPMENT OF THE NUMERICAL MODEL 44
   4.1. Class CsubDrainView 44
      4.1.1. CSubDrainView::OnProjectSubmergedPreprocessor() 44
      4.1.2. CSubDrainView::OnProjectRunModel() 45
      4.1.3. CSubDrainView::OnPojectRunRadialModel() 46
      4.1.4. CSubDrainView::OnPojectRunOneIteration() 46
      4.1.5. CSubDrainView::OnPojectViewModel() 46
      4.1.6. CSubDrainView::OnPojectViewRadialModel() 46
      4.1.7. CSubDrainView::OnResistanceFittingFunction() 46
      4.1.8. CSubDrainView::OnProjectContinueAfterFitting() 47
      4.1.9. OnDraw() 47

   4.2. Class setup 47
      4.2.1. SetRegsGrid() 47

TABLE OF CONTENTS ---*Continued*

Page

4.2.2. Trigrid() 47
4.2.3. SetIniSubCondition() 48
4.2.4. find_radial_elements() 48
4.2.5. average_height() 48
4.2.6. adjust_height() 48
4.2.7. drain_flux() 48
4.2.8. calculate_transmissivity() 48
4.2.9. solve_matrix() 48
4.2.10. water_table_flux() 49
4.2.11. water_table_movement() 49
4.2.12. KirkhamSolution() 49

4.3. Other Classes 49
4.3.1. Class element_matrix 49
4.3.2. Class vecti 50
4.3.3. Class vectd 50
4.3.4. Class vectf 50
4.3.5. Class matrixi 50
4.3.6. Class matrixd 50
4.3.7. Class matrixf 50
4.3.8. Class d3_matrixi 50

5. MODEL VALIDATION 51
5.1. Investigated Cases 52
5.1.1. Results and Discussion 53
5.1.1.1. Case 1 53
5.1.1.2. Case 2 56
5.1.1.3. Case 3 58

6. PREDICTING WATER TABLE FALL AND DISCHARGE 63
6.1. Fitting Resistance Function 63

6.2. Example 64
6.2.1. The Problem 64
6.2.2. Solution 65

6.3. Analysis of Results 74

7. CONCLUSIONS 76

TABLE OF CONTENTS ---*Continued*

Page

APPENDIX A DRAIN SYSTEM EQUATIONS TO PREDICT DISCHARGE AND WATER TABLE FALL 77

APPENDIX B CODE: MAIN FUNCTIONS 82

REFERENCES 123

LIST OF FIGURES

Figure		Page
1	Interpolation function N_m for node m.	30
2	Variables in the Green's Theorem.	32
3	Control volume used in the finite element method.	35
4	Variables in the water table position differential equation.	38
5	Terms used in the integration of Brook-Corey equation over the control volume at a horizontal distance, x, from drain.	40
6	The initial thirteen regions.	45
7	Kirkham variables for solution to the Laplace equation for subsurface drainage with steady state recharge.	51
8a	Kirkham and SubDrain water table elevation vs. distance from the drain: L= 60 m, R =0.000025 m/h, K = 0.003 m/h, and d = 3 m; alpha = drainage transfer coefficient.	54
8b	Kirkham and SubDrain water table elevation vs. distance near the drain: L = 60 m, R = 0.000025 m/h, K = 0.003 m/h, and d = 3 m; alpha = drainage transfer coefficient.	55
8c	Water table elevation at midpoint between drains vs. drainage transfer coefficient, alpha: L = 60 m, R = 0.000025 m/h, K = 0.003 m/h, and d = 3 m.	55
9a	Kirkham and SubDrain water table elevation vs. distance from the drain: L = 60 m, R = 0.00005 m/h, K = 0.0085 m/h, and d = 3 m; alpha = drainage transfer coefficient.	56
9b	Kirkham and SubDrain water table elevation vs. distance near the drain: L = 60 m, R = 0.00005 m/h, K = 0.0085 m/h, and d = 3 m; alpha = drainage transfer coefficient.	57
9c	Water table elevation at midpoint between drains vs. drainage transfer coefficient, alpha: L = 60 m, R =0.00005 m/h, K = 0.0085 m/h, and d = 3 m.	57

LIST OF FIGURES --Continued

Figure		Page
10a	Kirkham and SubDrain water table elevation vs. distance from the drain: L = 60 m, R = 0.0003 m/h, K = 0.05 m/h, and d = 3 m; alpha = drainage transfer coefficient.	58
10b	Kirkham and SubDrain water table elevation vs. distance near the drain: L = 60 m, R = 0.0003 m/h, K = 0.05 m/h, and d = 3 m; alpha = drainage transfer coefficient.	59
10c	Water table elevation at midpoint between drains vs. drainage transfer coefficient, α: L = 60 m, R =0.00005 m/hr, K = 0.0085 m/hr, and d = 3 m.	59
11	Relationship between the differences in maximum water table heights and wet entry perimeters for Kirkham and SubDrain method.	61
12	Drain system characteristics used in the example.	65
13	Grid for the initial condition in the example.	66
14	Resistance function, $\Omega_f(x)$, calculated with the least squares regression and ratio h(x)/R obtained with the finite element method.	67
15	Water table heights calculated with Eq. 36 and with the finite element method at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x = 4.81 m), and at the middle between drains (x = 20 m).	68
16	Water table height differences calculated with Eq. 36 (h_r) and with the finite element method (h_f) at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x = 4.81 m), and at the middle between drains (x = 20 m).	69
17	Fluxes calculated with Eq. 37 (f_r) and with the finite element method (f_f) at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x = 4.81 m), and at the middle between drains (x = 20 m).	70
18	Flux differences calculated with Eq. 37 (f_r) and with the finite element method (f_f) at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x = 4.81 m), and at the middle between drains (x = 20 m).	71

LIST OF FIGURES --Continued

Figure		Page
19	Total discharge per 1 m of drain vs. time, calculated with Eq. 38 and the finite element method.	73
20	Percent difference of total discharge obtained with Eq. 38 (Q_r) and the finite element method (Q_f).	73
A.1	Frozen stream tubes and variables of the water table fall equation.	78

LIST OF TABLES

Table		Page
Table 5.1	Soil types, recharge rates, and drain transfer coefficients for comparison of Kirkham analytic solution with SubDrain solution. . .	52
Table 5.2	Comparison of Kirkham and SubDrain maximum water table height; fixed values of potential head in the drain were used.	60
Table 5.3	Comparison of drain discharge and total recharge; drain discharge was obtained with the drain_flux() function and total recharge with the water_table_flux() function.	62
Table 6.1	Comparison of water table heights (m) and fluxes (m/hr) calculated with the equations 36 and 37, and with the finite element method at three nodes.	72

ABSTRACT

A free water surface finite element model was developed. The method was implemented with the Galerkin approach to solve the Laplace equation in the saturated region. It was developed in the object oriented Visual C^{++} computer language to permit easy update and drawing of the adaptive mesh. For each time step, the new water table position was calculated based on flux across the water table, a Brooks-Corey equation mass balance for the unsaturated region, and an equation that calculates water table position for the saturated region. An equation was developed to calculate a drainage transfer coefficient, α, based on percentage of perforated area in the drain tube wall. The drainage transfer coefficient was incorporated into the finite element model as a Fourier boundary condition. To validate the finite element model, its results were compared with the Kirkham equation results for steady state recharge of three subsurface drainage systems.

The finite element model was used to calibrate a semi-analytical frozen stream tube model for subsurface drainage of heterogeneous soils. The first step in the calibration procedure is to run the finite element model for steady state recharge and calculate the water table height divided by recharge rate (the stream tube resistance to flow) as a function of distance between drains. Least squares regression is used to fit a polynomial logarithmic equation, called the resistance function, to the stream tube resistance to flow vs. distance from the drain curve. A differential equation based on the principle of conservation of mass and application of Darcy's law to the frozen stream tube

was solved to obtain an equation that calculates stream tube flow rate and final water table elevation as a function of the resistance function and initial water table elevation.

An example was developed for a non-homogeneous subsurface drainage system to illustrate the use of the semi-analytical model to predict water table fall and discharge.

CHAPTER 1

INTRODUCTION

The main purpose of drainage of agricultural lands in humid climates is removal of excess water to improve the profitability of farming the land. Throughout the world, experience has shown that successful irrigation requires adequate drainage; land can not be permanently irrigated when natural drainage is inadequate and artificial drainage is not provided. Also, in irrigated areas, a second function of drainage is to remove salts and to provide control over the quality of the discharged drain water.

The fundamental problem in the design and management of subsurface drainage systems is to know the water table levels, fluxes, and discharges with time.

The methods to solve this problem can be subdivided into analytical methods, numerical methods, physical model, and analogs.

Analytical solutions of differential equations are exact mathematical models of physical phenomena; nevertheless, for drainage, analytic solutions are often not possible or accurate because of geometric complexity or soil anisotropy. Physical models and analogs have been used to model drainage or verify analytical solutions that were based on simplifying assumptions (Bear, 1972). With the advent of computers, numerical methods can be used to obtain a model of complex soils or configurations. Furthermore, in some situations, numerical models can be utilized to attain quasi-analytical solutions.

Previous analytical drainage models predicted the water table height with time in flat, rectangular, homogeneous or two layer drain systems, but they did not take seepage into account.

Numerical drainage models are more robust than analytic models, but they have the limitation of high computation time for modeling water systems with multiple drains. For current hydrologic models, equations that rapidly predict the water table level and discharge with time in non-homogeneous subdrain systems would be useful.

This work combines the strength of numerical models with the speed of analytic solutions. Semi-analytical solutions to obtain the water table position, fluxes and discharge with time in drainage systems with more than two layers are developed. A finite element model that considers seepage into the drain is coded to obtain the resistance function required in the semi-analytical solutions. Furthermore, this model can simulate submerged and unsubmerged drain conditions.

The general objective of this work is to develop equations that predict with time the water table profile, fluxes and discharges for layered drain systems.

The particular objectives are:

1. Generate a numerical model to obtain with time the water table position, fluxes and discharges for submerged and unsubmerged subdrain systems, given recharge, and physical and geometric characteristics of the system.
2. Compare numerical results with Kirkham's analytical solution.

3. Develop semi-analytical solutions to predict the water table profile, fluxes and discharge over time in drain systems.
4. Generate a hydraulic function (resistance function) based on numerical results and the frozen stream tube theory.

The dissertation is broken into the following sections:

Chapter 2 is a literature review of the problem: to obtain the water table position and discharges in subsurface drainage systems.

Chapter 3 provides the theoretical background of saturated water flow, Galerkin method, water table position differential equation, entry losses in the subsurface drain, water table fall equation, fluxes equation, and discharge equation. This theory is required to develop the numerical method: finite element method, and the implementation of the semi-analytical equation.

Chapter 4 describes the theoretical development of the numerical model and of the numerical implementation of the semi-analytical solutions.

Chapter 5 compares the new model to the Kirkham analytical equation.

Chapter 6 describes the development of the total resistance function, $\Omega(x)$, and it also shows an example with an application of the semi-analytical equations that predict water table fall, fluxes, and discharge.

Conclusions are given in Chapter 7.

Appendix A includes the equation derivation for prediction of water table fall and discharge based on the frozen stream tube theory.

Appendix B is the code of the main functions of the developed model.

The model of tile drainage that was developed in this work is called SubDrain.

CHAPTER 2

LITERATURE REVIEW

The water table level, fluxes, and discharges for subsurface drainage has been calculated with several approaches: analytical methods, numerical methods, physical models, and analogs.

2.1 Analytical Solutions

By simplifying the governing differential equations, the geometry, or the properties of the soils, extraordinary analytical models of drainage have been derived by many researchers.

Colding in 1872, cited by van der Ploeg et al. (1997), was the first to develop the ellipse equation; this equation is based on the Dupuit-Forchheimer theory, and it describes, for steady state recharge, the shape of the water table between parallel drains for a homogeneous soil underlain by a horizontal impervious barrier.

Hooghoudt, in 1940, developed a formula for steady state drainage that calculates midpoint water table height based on drain spacing, hydraulic conductivity, depth of water in the ditch, and steady state recharge in systems with parallel open ditches reaching the impermeable substratum. He also modified the equation for parallel pipe drains above the substratum (Smedesma and Rycroft, 1983).

Kirkham (1958) developed analytical solutions for streamlines and heights of the water table for parallel tile drains with homogeneous soil underlain by an impermeable layer and steady rainfall.

Bouwer and van Schilfgaarde (1962) developed a simplified procedure for predicting the rate of water table fall midway between drains in tile drained and ditch drained land. The procedure is based on the steady state theory and the assumption that the shape of water table does not change while it falls. They introduced the "C factor" to account for the non-uniform flux per unit area of water table if the water table shape changes during its fall.

Van Schilfgaarde (1963) proposed a technique to determine the depth and spacing of subsurface drains; this technique is based on the soil properties and the specified rate of drawdown of the water table. His procedure is applicable for homogeneous flat systems with parallel drains and an impervious barrier below the drain. To obtain his formula he used the Dupuit-Forchheimer assumption, and the Houghoudt correction for the effect of convergence of the flow close to drains.

Applying conformal mapping, Hammad (1963) obtained two equations for steady state drainage that calculate the discharge and the maximum height of the water table based on spacing, tile diameter, coefficient of permeability, and depth of the impermeable stratum. A homogeneous isotropic soil and a horizontal impermeable bed were assumed. Furthermore, starting with an initial steady state position, he developed an implicit equation that relates the spacing of drains, time increment, initial and final maximum water table height, physical soil characteristics, depth of impermeable bed, pipe diameter, and evaporation. Because of the assumption of small slopes in the water table, the solution gives reliable results only when drain is submerged.

Using fixed streamlines and the water movement along the streamtubes to parallel tile drains, Kirkham (1964) calculated the rate of water table fall in homogeneous soil after steady rainfall ceased. This approach also was used by Jury (1975) to calculate solute travel time to tile drains for a parallel tile arrangement in a flat landscape.

Solving the Boussinesq equation, water table equilibrium with steady rainfall on sloping land was calculated by Schmid and Luthin (1964); in this work, they assumed parallel equally spaced ditch drains based on an impermeable layer. They prepared a nomograph for calculation of ditch spacing as a function of steady rainfall intensity, soil hydraulic conductivity and maximum height of the water table above the impermeable layer.

The Boussinesq equation is

$$S\frac{\partial h}{\partial t} = K\frac{\partial}{\partial x}\left(h\frac{\partial h}{\partial x}\right) \qquad [1]$$

where K is the hydraulic conductivity, S is the specific yield, h is the water depth, x is the horizontal coordinate, and t is time.

Towner in 1975 obtained an analytical solution of Childs' differential equation (Childs, 1971), and showed that his solution is in better agreement with the experimental results of Guitjens and Luthin (1965) than that obtained with Dupuit's assumption (vertical equipotentials).

Ram and Chauhan (1987) derived an analytical model of the rising water table in response to constant replenishment in an aquifer lying over a mildly sloping impervious stratum; they based their solution on the Boussinesq equation, and assumed homogeneous

isotropic soil, and parallel drains perpendicular to the slope of the impermeable layer. The solution, which is an infinite series, was verified with a Hele-Shaw model. The main disadvantage of this solution is that it is valid only for a drain resting on an impermeable bed.

Yu and Konhya (1992) generated streamlines to drainage tiles with a boundary model solution to the Laplace equation. The model predicted water table position, head loss at the drain as well as fluxes and potentials along the boundary for steady state drainage. The model had the advantage of only defining boundary elements while at the same time handling variable hydraulic conductivity with depth.

Based on function analysis, Toksööz and Kirkham obtained an equation that relates the maximum water height with the soil and hydrologic parameters of two-layer flat subdrain systems; steady state recharge and no seepage were considered in their analysis (van der Ploeg et al. 1999).

2.2 Physical Models and Experiments

Guitjens and Luthin (1965) conducted experiments with a Hele-Shaw apparatus to appraise the error due to the Dupuit-Forchheimer assumption on sloping soils. The Hele-Shaw model validated the Schmid and Luthin (1964) model for up to 30% slope for an equilibrium water table with steady rainfall.

Asseed and Kirkham (1966) built a laboratory model of subsurface drainage. Change in depth of a shallow porous medium markedly influenced drainage. A critical

depth occurred when the barrier was at a depth equal to one fifth of the drain spacing; below that depth, the depth of the barrier had negligible influence on rate of fall of the water table and on drain discharge.

Luthin and Guitjens (1967) developed dimensionless graphs for analysis of a transient falling water table on sloped land; the graphs were developed with data from a Hele-Shaw model of parallel open ditches on sloped land. Because they proved that the laws of similitude hold for transient and steady state cases, their dimensionless graphs can be applied to drainage system design if the analytic results obtained in 1964 by Schmid and Luthin are used to define the initial condition. Because their results also showed that the rate of water table fall is almost independent of slope, up to 30% slope if the ditches penetrate to the impermeable layer, they proposed that flat land drainage theory can be applied to sloping land without much error.

Mohammad and Skaggs (1983) conducted experiments to determine the drainage transfer coefficient, α, for 4 in diameter corrugated plastic drains; they found that α increased with fraction of entry area in the drain pipe from less than 1 h^{-1} for an entry area of 1.8 cm^2/m to 33 h^{-1} for an entry area of 182 cm^2/m. There was not a linear relationship between entry area and α. The coefficient α is the ratio of the output velocity in the drain and the head losses through it.

2.3 Numerical Models

Kirkham and Gaskell (1950) used a finite difference model to calculate the rate of fall of the water table when the soil is initially saturated to the surface. They assumed

homogeneous soil, horizontal surface, an impermeable layer below the drain, and equally spaced tiles.

Cited by Smedema and Rycroft (1983), Zeeuw and Hellinga (1958) used the Hooghoudt formula to develop, for nonsteady state drainage, two iterative exponential equations that obtain the discharge and the midpoint water table height as a function of the reaction factor, time increment, recharge, and drainable pore space. The reaction factor - an index of the intensity with which the drain discharge responds to changes in the recharge, includes the physical and geometric characteristics of the drain system.

Moody (1966) presented graphic dimensionless solutions that relate spacing of parallel pipe drains, time, maximum water height, depth of impermeable layer, soil physical properties, volume of water removed, and rate of discharge, in homogeneous flat drain systems. The graphs were obtained by solving a nonlinear partial differential equation with the finite difference method. Since this equation was based on the simplifying Dupuit assumption, he also gave an approximate formula to consider radial losses.

Assuming equipotentials perpendicular to the bed, Childs (1971); obtained a water table height differential equation in drain systems with sloping impermeable beds, equidistant parallel ditch drains across the slope, and constant surface recharge. He solved this equation with numerical methods and compared the height with that obtained with Dupuit's assumption. He concluded that the maximum height is only affected for small

rates and high slopes, and the position of the maximum height is shifted downhill but not enough for practical concern.

Using the Galerkin technique, Dass and Morel-Seytoux (1974) solved the nonlinear Boussinesq equation for three kinds of initial water profiles between drains in flat homogeneous systems; the radial flow close to the drain was not taken account in their analysis.

With a finite element model, Merva et al. (1981) predicted the water table position with time in two-dimensional drain systems that had horizontal multi-layered soils; in this model, the Laplace equation was solved for each time step, and the next water table position was obtained by equating the volume of water flowing at each node to the volume of water removed from the soil. The model was validated with field data.

Fipps et al. (1986) tested four methods for representing a drain in the finite element solution of the Richards equation: (1) a hole surrounded with a dense mesh equal in size to the effective radius, (2) a sink at a nodal point with the flux specified using the Kirkham equation developed for a ponded surface, (3) a single node with the hydraulic head specified, and (4) elements around a single node with their hydraulic conductivities modified with the Vimoke and Taylor adjustment factors. The methods were validated with the analytic solution of Kirkham for the case of parallel full drains, homogeneous soil, and surface ponding.

Fipps (1989) developed and tested an approximate method for determining transient drainage that combined the Houghoudt equation, a radial flow equation, and a

variable porosity equation. His method was found to be in close agreement with finite element solutions of the Richards equation when the initial water table is elliptical, and after the transition period for the case of an initial flat water table. Only submerged full drain conditions were investigated.

Fipps and Skaggs (1989) investigated hillside drainage using numerical solutions to the two-dimensional Richards equation. The influence of slope on the drain discharge and water table height at the center region was examined. Parallel drainage and steady rain was assumed. The results suggest that small slopes (≤ 0.15 m/m) had little effect on drain flow and water table depths in the center of the region between drains. Their study indicated that the drainage theory developed for flat land could be used for parallel drains on slopes less than 0.15 m/m.

Scotter et al. (1990) developed a simple numerical solution for transient soil water flow to a mole drain. Dupuit-Forchheimer flow below the water table and vertical potential equilibrium above the water table were assumed.

Using variable drainable porosity, the Boussinesq equation, and a finite difference scheme, Pandey et al. (1992) obtained the drawdown of the water table between parallel drains in a homogeneous soil underlain by an impermeable layer. Based on their experimental work, they discovered that variable drainable porosity in the numerical solution increased the accuracy of the prediction of the water table fall compared with a constant value for drainable porosity.

There are several models that calculate solute movement to parallel tile drains. "Drainmod-N" is a computer simulation model for predicting nitrogen transport, plant uptake, and transformations in artificially drained soils (Breve et al., 1992). This model uses "Drainmod", a tile drainage model based on the Hooghoudt equation to calculate vertical soil water fluxes in the profile and water table position. The model was tested by comparison with a more complex numerical model and the results were in good agreement.

Ŝimůnek et al. (1994) developed a code (SWMS_2D) for simulating water flow and solute transport in two-dimensional variably saturated media. Because the model uses the finite element method to solve the Richards equation and the transport equation, anisotropy and heterogeneity can be simulated with irregular boundaries. However, drains only can be represented with single grid points having zero pressure, which correspond to empty drains. Because drains behave as nodal sink/sources with zero recharge when located in the unsaturated zone, they can not be simulated accurately in the partially submerged condition.

CHAPTER 3

THEORETICAL BACKGROUND

In this chapter, the equations and theoretical concepts used in the finite element method and the functions to predict the water table position and discharge of the SubDrain model are described.

3.1 Finite Element Method

The finite element method has two particular characteristics: it uses an integral formulation to obtain a system of algebraic equations, and it utilizes piecewise smooth functions for approximating the unknowns.

The advantages of the finite element method are that it can be easily applied when having (1) different materials, (2) mixed boundary conditions, (3) steady and non-steady state, and (4) nonlinear material properties (Segerlind, 1984).

The finite element method includes the following five basic steps: (1) discretization of the region, (2) specification of the approximation equation in terms of the unknown nodal values, (3) developing the system of equations, (4) solving the system of equations, and (5) calculating quantities of interest -- gradients, velocities, and discharges (Segerlind, 1984).

3.1.1 Governing Differential Equation

The governing differential equation for steady state water flow through saturated soil is the Laplace equation; this equation for two dimensions is

$$K_x \frac{\partial^2 H}{\partial x^2} + K_z \frac{\partial^2 H}{\partial z^2} = 0 \qquad [2]$$

Where H is the total head (total potential). The most common practice for subsurface drainage modeling is to fix the datum for total potential at the center of the tile drain. Parameters, K_x and K_z, are the permeability coefficients in the x and z direction, respectively.

The most accurate solution of a governing equation such as the Laplace equation is an analytical solution; however, there are situations where this approach is not possible because the control volume is irregular or includes heterogeneity. In these cases, numerical methods can approximate the solution. The finite element method is one of the most popular numerical methods.

3.1.2 Galerkin's Method

In Galerkin's approach for the finite element method, a weighting function is defined for each nodal unknown, and the weighted residual integral is evaluated at each node. A solution $\underline{H}(x,z)$ is used to approximate the exact solution H(x,z); this approximate solution has the form

$$\underline{H}(x,z) = \sum_{m=1}^{NN} \underline{H}_m N_m(x,z) \qquad [3]$$

where

$\underline{H}_m$ = $\underline{H}$ at node m,

N_m = piecewise linear Lagrange interpolation function; 1 at node m and 0 at adjacent nodes (fig. 1),

NN = total number of nodes.

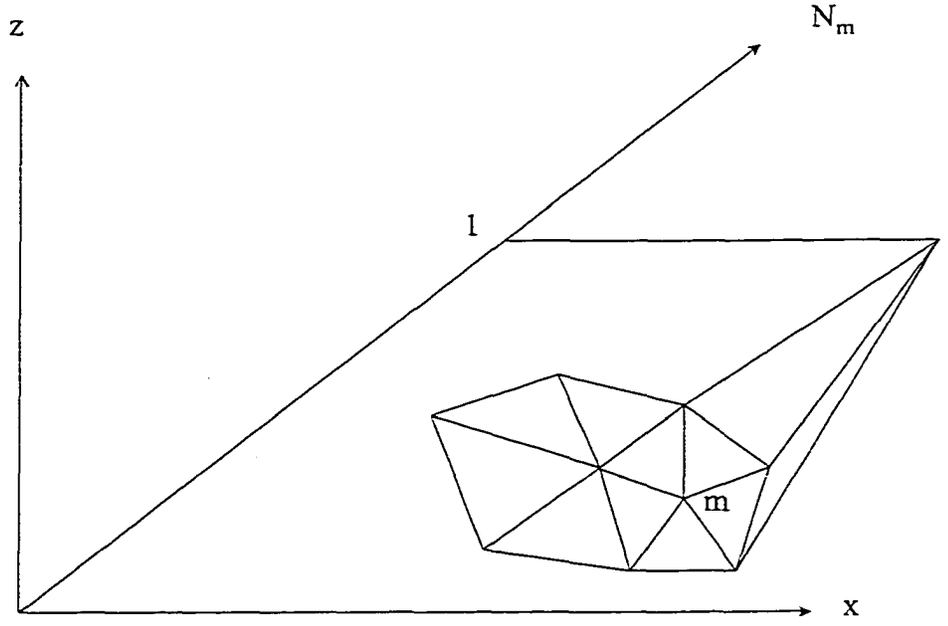


Fig. 1 Interpolation function N_m for node m.

If we substitute the approximate solution, Eq. 3, into the Laplace equation, the solution is not exact, and there is a residual error:

$$K_x \frac{\partial H}{\partial x} + K_z \frac{\partial H}{\partial z} = E(x,z) \qquad [4]$$

To keep this error as small as possible, at each node the integral of the error over the elements around the node weighted by the interpolation function N_n is forced to be zero:

$$\iint_{A_n} E(x,z) N_n(x,z) dA = 0 \qquad [5]$$

If the residual, R_n is defined as the negative of this integral we obtain the equation

$$R_n = -\iint_{A_n} E(x,z) N_n(x,z) dA = 0 \qquad [6]$$

In this way, a system of equations is derived and solved for $\underline{H}$ at all nodes. Each node has a particular approximating function, N_n, and the number of approximating equations is the same as the number of nodes. The system of equations is set in a matrix equation and solved in order to obtain the unknowns.

Eq. 6 can be written in terms of the contribution of each element around node n:

$$R_n = \sum_{e=1}^{l} R_n^e = -\sum_{e=1}^{l} \iint_{A^e} E(x,z) N_n^e(x,z) dA = 0 \qquad [7]$$

where

l = number of elements around n

e = element number

On the other hand, since each element contributes to the equations of all its nodes, the element contribution to the system of equations can be expressed by

$$\{R^e\} = -\iint_{A^e} [N^e]^T \left(K_x \frac{\partial^2 \underline{H}}{\partial x^2} + K_z \frac{\partial^2 \underline{H}}{\partial z^2} \right) dA \qquad [8]$$

where

$\{R^e\}$ = column vector of the element contributions to the system of equations.

$[N^e]$ = row vector containing the element interpolation functions.

A^e = area of the element e.

If the approximate solution, $\underline{H}(x,z)$, is piecewise linear the second derivatives and the components of $\{R^e\}$ will be zero; to avoid this, Green's theorem should be applied. After applying Green's theorem for a given element, (see fig. 2), we obtain

$$\{R^e\} = -\int_\Gamma [N^e]^T \left(K_x \frac{\partial \underline{\underline{H}}}{\partial x} \cos\theta + K_z \frac{\partial \underline{\underline{H}}}{\partial z} \sin\theta \right) d\Gamma + \left(\int_{A^e} \left(K_x \frac{\partial [N^e]^T}{\partial x} \frac{\partial [N^e]}{\partial x} + K_z \frac{\partial [N^e]^T}{\partial z} \frac{\partial [N^e]}{\partial z} \right) dA \right) \{\underline{\underline{H}}^e\} \quad [9]$$

or

$$\{R^e\} = \{I^e\} + [K^e]\{\underline{\underline{H}}^e\} \quad [10]$$

where

$$\{I^e\} = -\int_\Gamma [N^e]^T \left(K_x \frac{\partial \underline{\underline{H}}}{\partial x} \cos\theta + K_z \frac{\partial \underline{\underline{H}}}{\partial z} \sin\theta \right) d\Gamma \quad [11]$$

and

$$\{K^e\} = \int_{A^e} \left(K_x \frac{\partial [N^e]^T}{\partial x} \frac{\partial [N^e]}{\partial x} + K_z \frac{\partial [N^e]^T}{\partial z} \frac{\partial [N^e]}{\partial z} \right) dA \quad [12]$$

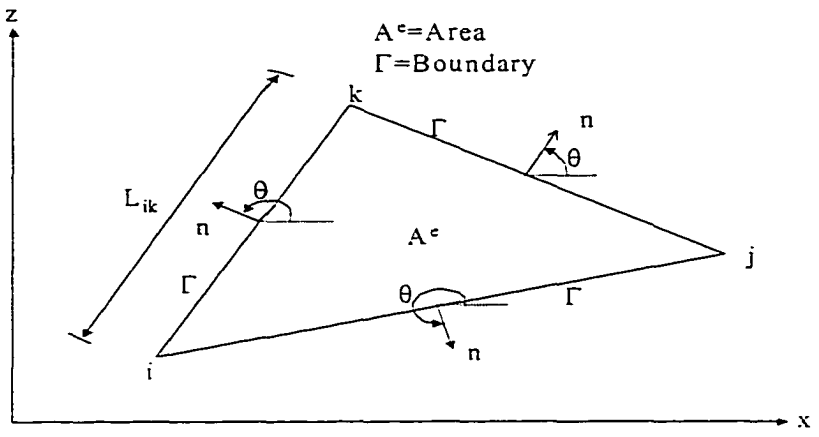


Fig. 2 Variables in the Green's Theorem.

For triangular elements (Segerlind, 1984):

$$\left[K^e\right]=\frac{K_x}{4A^e}\begin{bmatrix} b_i^2 & b_ib_j & b_ib_k \\ b_ib_j & b_j^2 & b_jb_k \\ b_ib_k & b_jb_k & b_k^2 \end{bmatrix}+\frac{K_z}{4A^e}\begin{bmatrix} c_i^2 & c_ic_j & c_ic_k \\ c_ic_j & c_j^2 & c_jc_k \\ c_ic_k & c_jc_k & c_k^2 \end{bmatrix} \qquad [13]$$

where

$b_i = z_j - z_k$

$b_j = z_k - z_i$

$b_k = z_i - z_j$

$c_i = x_k - x_j$

$c_j = x_i - x_k$

$c_k = x_j - x_i$

Eq. 11 is not applied to sides common to two elements because the values in one element are the same as the values in the other element but with opposite sign; the integration is only done in boundary sides.

If in a boundary side the boundary condition is of the type

$$K_x\frac{\partial H}{\partial x}\cos\theta + K_z\frac{\partial H}{\partial x}\sin\theta = -MH_b + S \qquad [14]$$

where M and S are constants and H_b is the unknown total head in that side, then for triangular elements $\{I^e\}$ is given by (Segerlind, 1984):

$$\left\{I^e\right\}=\left[K_M^e\right]\left\{H^e\right\}-\left\{f^e\right\} \qquad [15]$$

where

$$\left[K_M^e\right]=\frac{ML_{ij}}{6}\begin{bmatrix} 2 & 1 & 0 \\ 1 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \text{ for side ij} \qquad [16]$$

or

$$\left[K_M^e\right] = \frac{ML_{jk}}{6}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 2 & 1 \\ 0 & 1 & 2 \end{bmatrix} \text{ for side jk} \qquad [17]$$

or

$$\left[K_M^e\right] = \frac{ML_{ik}}{6}\begin{bmatrix} 2 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 2 \end{bmatrix} \text{ for side ik} \qquad [18]$$

and

$$\left\{f_s^e\right\} = \frac{SL_{ij}}{2}\begin{Bmatrix} 1 \\ 1 \\ 0 \end{Bmatrix}, \quad \left\{f_s^e\right\} = \frac{SL_{jk}}{2}\begin{Bmatrix} 0 \\ 1 \\ 1 \end{Bmatrix}, \quad \text{and } \left\{f_s^e\right\} = \frac{SL_{ik}}{2}\begin{Bmatrix} 1 \\ 0 \\ 1 \end{Bmatrix} \qquad [19]$$

for sides ij, jk, and ik respectively.

3.1.3 Boundary Conditions

The control volume for this model is shown in fig. 3. In the vertical direction, the volume extends from the water table to the impermeable layer. Thus, element positions are moved at each time step. In the horizontal direction, the control volume extends from the drain to the midpoint between drains. The drain is located above the impermeable layer, and a datum is located at the vertical midpoint of the drain.

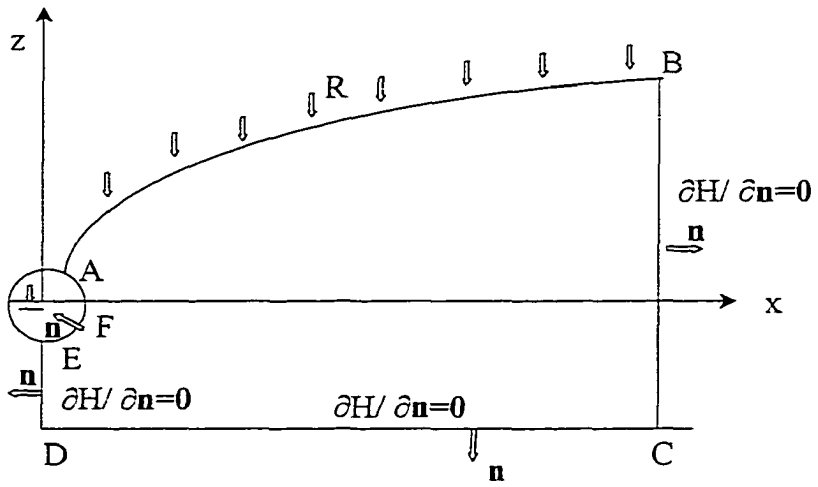


Fig. 3 Control volume used in the finite element method.

The boundary conditions required to solve Eq. 2 in this particular problem are (1) no flow across the side or lower boundaries except for flow into the drain, (2) total potential on the lower half of the drain boundary equals elevation of the center of the drain (drain flows half full), (3) total potential on the upper half of the drain equals elevation, and (4) total potential of nodes on the water table equals elevation.

Boundary conditions can be expressed with a mathematical model. There are three types of mathematical models of boundary conditions (Marsily, 1986): prescribed boundary conditions, Neumann's conditions, and Fourier's boundary condition.

Prescribed head boundaries, Dirichlet's condition, are imposed where the total head on the boundary is independent of the flow - the potential at the boundary is

specified with time. In our model, for each time step, the total potentials at the water table nodes equal to their elevations.

Prescribed flux boundaries, Neumann's condition, are imposed where the flux on the boundary is independent of potential, and the first derivative of the potential is prescribed

$$\frac{\partial H}{\partial \mathbf{n}} = c \qquad [20]$$

For the drain system model, the flux across the lower and side boundaries is zero, sides BC, CD, and DE of fig. 3. Thus, Eq. 20, Neumann's condition, is set equal to zero for those boundaries.

$$\frac{\partial H}{\partial x} = 0 \text{ for sides, } \frac{\partial H}{\partial z} = 0 \text{ for lower boundary} \qquad [21]$$

Fourier's condition is imposed where the normal flux across the boundary is a linear function of the head at the same point, i.e.

$$\frac{\partial H}{\partial \mathbf{n}} = c_1 H + c_2 \qquad [22]$$

This boundary condition is used at the entrance to the drain where flow through the drain is dependent on the drain resistance to flow and the potential difference across the boundary of the drain.

$$-KA\frac{\partial H}{\partial \mathbf{n}} = \alpha A\left(H_{in} - H_{out}\right) \qquad [23]$$

where

H_{in} = potential on the soil side of the drain, m,

H_{out} = potential within the drain, m,

A = Area, m^2.

α = drainage transfer coefficient, h^{-1}.

If Eq. 23 is rearranged we have an equation similar to Eq. 14

$$K\frac{\partial H}{\partial \mathbf{n}} = \alpha H_{out} - \alpha H_{in} \qquad [24]$$

Because the datum is defined at the center of the drain, and the drain is half filled with water, $H_{out} = 0$ for nodes below the center of the drain, and H_{out} = node elevation, z, for nodes above the center of the drain.

The recharge at the water table surface is taken into account to obtain the new position of the water table after the Laplace equation is solved in each time step.

3.1.4 Adaptive Mesh

Finite element models require a mesh. There are many mesh generation programs. Trigrid, one such mesh generation program developed by John Nieber (private communication in 1993) generates linear triangular elements. Trigrid was incorporated into the SubDrain finite element component. At each time step the nodes are moved to cover just the new saturated region.

3.1.5 Water Table Position Differential Equation and Drainable Porosity Equation

The change of the water table height with time is a function of the water table slope, hydraulic gradient, permeability coefficient, and drainable porosity (Waller and Jaynes, 1993); with reference to fig. 4 the relationship is given by:

$$\frac{\partial h}{\partial t} = \frac{K\left(\frac{\partial H}{\partial z} - \frac{\partial H}{\partial x}\frac{\partial h}{\partial x}\right)}{f} \qquad [25]$$

where:

f	=	drainable porosity, m/m,
h	=	water table height above the drain center, m,
H	=	total head = P + z, m,
K	=	permeability coefficient, m/h.
P	=	pressure head, m,
x	=	horizontal distance from drain, m,
z	=	vertical distance above drain center, m.

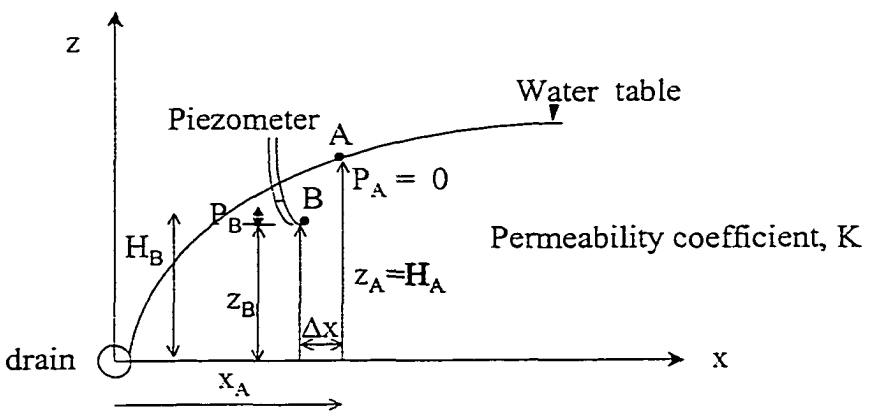


Fig. 4 Variables in the water table position differential equation.

Drainable porosity f, is the ratio between the depth of water removed from the soil and the change in the water table height; it can be derived by applying the Brooks-Corey equation to a control region that includes the region above the water table. To apply this equation in the study of the water table fall due to drainage, it is convenient to define the control volume as the volume of soil above the drain center in a given cross-sectional area between drains.

The Brooks-Corey equation relates water content with pressure head as follows

$$\theta_e = \frac{\theta - \theta_r}{\theta_s - \theta_r} = \left(\frac{P_b}{P}\right)^{\lambda} \qquad \text{if } P < P_b$$
$$\theta = \theta_s \qquad \text{if } P => P_b \qquad [26]$$

or

$$\theta = \theta_r + (\theta_s - \theta_r)\left(\frac{P_b}{P}\right)^{\lambda} \qquad \text{if } P < P_b$$
$$\theta = \theta_s \qquad \text{if } P => P_b \qquad [27]$$

where:

P_b = bubbling pressure head, m,

P = soil water pressure head, m,

λ = coefficient,

θ = water content, m/m,

θ_e = effective water content, m/m,

θ_r = residual water content, m/m,

θ_s = saturated water content, m/m.

The total depth of water in the control volume at a horizontal distance, x, from drain can be obtained integrating the water content from the drain center level to the surface (fig. 5)

$$d_w(x) = V_w/A = \frac{1}{A}\int_0^d A\theta dz = \int_0^d \theta dz \qquad [28]$$

where:

A	=	cross area of the soil column, m^2,
$d_w(x)$	=	depth of water at x, m,
V_w	=	volume of water in the soil column, m^3.

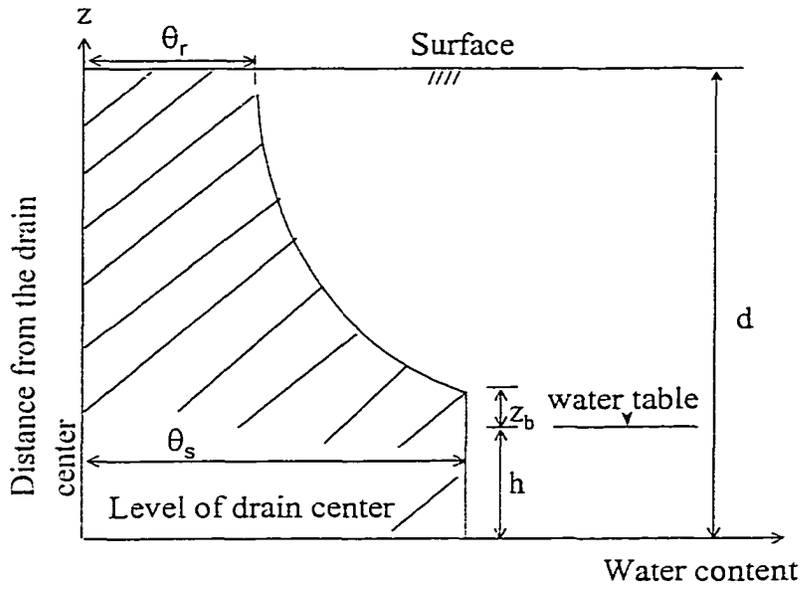


Fig. 5 Terms used in the integration of Brook-Corey equation over the control volume at a horizontal distance, x, from drain.

Eq. 27 can be expressed in terms of the height of the water table above the drain, substituting $P = -(z - h)$ if $z > (z_b + h)$ where $z_b = -P_b$, h = height of the water table above the drain center:

$$\theta = \theta_r + (\theta_s - \theta_r)\left(\frac{z_b}{z-h}\right)^{\lambda} \qquad \text{if } z > (z_b + h)$$
$$\theta = \theta_s \qquad \text{if } z <= (z_b + h) \qquad [29]$$

Substituting in Eq. 28 the value of θ given by Eq. 29

$$d_w(x) = \int_0^{h+zb} \theta_s dz + \int_{h+zb}^{d} \left[\theta_r + (\theta_s - \theta_r)\left(\frac{z_b}{z-h}\right)^{\lambda}\right] dz \qquad [30]$$

Integrating,

$$d_w(x) = \theta_s(h + z_b) + \theta_r(d - h - z_b) + \frac{(\theta_s - \theta_r) z_b^{\lambda}}{1-\lambda}\left((d-h)^{1-\lambda} - z_b^{1-\lambda}\right) \qquad [31]$$

Drainable porosity at x, f, is the derivative of d_w with respect to h

$$f = \frac{\partial d_w(x)}{\partial h} = (\theta_s - \theta_r)\left(1 - \left(\frac{z_b}{d-h}\right)^{\lambda}\right) \qquad [32]$$

Substituting f of Eq. 32 in Eq. 25 we obtain the water table elevation change as a function of the soil characteristics, the elevation of the water table, distance to the surface, and the gradient in the region of analysis:

$$\frac{\partial h}{\partial t} = \frac{K\left(\frac{\partial H}{\partial z} - \frac{\partial H}{\partial x}\frac{\partial h}{\partial x}\right)}{(\theta_s - \theta_r)\left(1 - \left(\frac{z_b}{d-h}\right)^{\lambda}\right)} \qquad [33]$$

3.1.6 Entry Losses

The head losses when the water flows toward and through the limited open area of the drain are called entry losses (Smedesma and Rycroft, 1983).

These losses h_e (m) are related with the drainage transfer coefficient α, h^{-1}:

$$h_e = \frac{Q}{A\alpha} \qquad [34]$$

where A = surface flow area of the tube, m^2, and Q = flow rate through the tube, m^3/h. For ideal tubes without losses, $\alpha = \infty$.

Using data from Table 3 from Mohammad and Skaggs (1983), the following relationship was found between the percentage of perforated area of the tube wall area, $A_p(\%)$, and the drain transfer coefficient, $\alpha(h^{-1})$:

$$\alpha = \frac{1}{-0.10035735 + \dfrac{0.314582243}{A_p(\%)^{0.5}}} \qquad [35]$$

$A_p(\%)$ varies from 0.05 to 5.8 (Mohammad and Skaggs, 1983).

3.2 Drain System Equations to Predict Water Table Fall and Discharge

For a given subsurface drainage system it is useful to have equations that give the water table fall and the discharge with time.

Assuming frozen stream tubes between two nodes over the water table, Eqs. 36, 37 and 38 give the water table position, flux, and total discharge with time:

$$h_1 = h_0 e^{-\Delta t/f\Omega(x)} + R_{av}\Omega(x)\left(1 - e^{-\Delta t/f\Omega(x)}\right) \qquad [36]$$

$$q_1 = q_0 e^{-\Delta t/f\Omega(x)} + R_{av}\left(1 - e^{-\Delta t/f\Omega(x)}\right) \qquad [37]$$

$$Q_1 = \sum_{i=1}^{n} q_{1i} \Delta x_i \qquad [38]$$

where q_1(flux) is the Darcy velocity at the beginning of the stream tube.

These equations are derived in Appendix A.

CHAPTER 4

THEORETICAL DEVELOPMENT OF THE NUMERICAL MODEL

SubDrain was developed in the object oriented Visual C^{++} computer lenguage. In this section, the main classes and functions utilized in SubDrain are described.

Since the program SubDrain applies the object-oriented approach, the functions presented in its menu can be run in any order after the object has been initialized with the function Submerged Preprocessor inside the class Project.

The main classes are CsubDrainView, and setup.

4.1 Class CsubDrainView

CsubDrainView class generates the object, operates the setup class functions to modify the properties of the object, draws the mesh and gradients, as well as writes function coefficients. Its main functions are OnProjectSubmergedPreprocessor(), OnProjectRunModel(), OnProjectRunRadialModel(), OnProjectRunOneIteration(), OnProjectViewModel(), OnProjectViewRadialModel(), OnResistanceFittingFunction(), OnProjectContinueAfterFitting(), and OnDraw().

4.1.1 CSubDrainView::OnProjectSubmergedPreprocessor()

Generates the grid for thirteen regions (see Fig. 6), subdivide these regions in triangle elements, and finds the boundary nodes and the elements around the drain. It uses the functions setup::SetRegs(), setup::Trigrid(), setup::SetIniSubCondition(), and setup:: find_radial_elements().

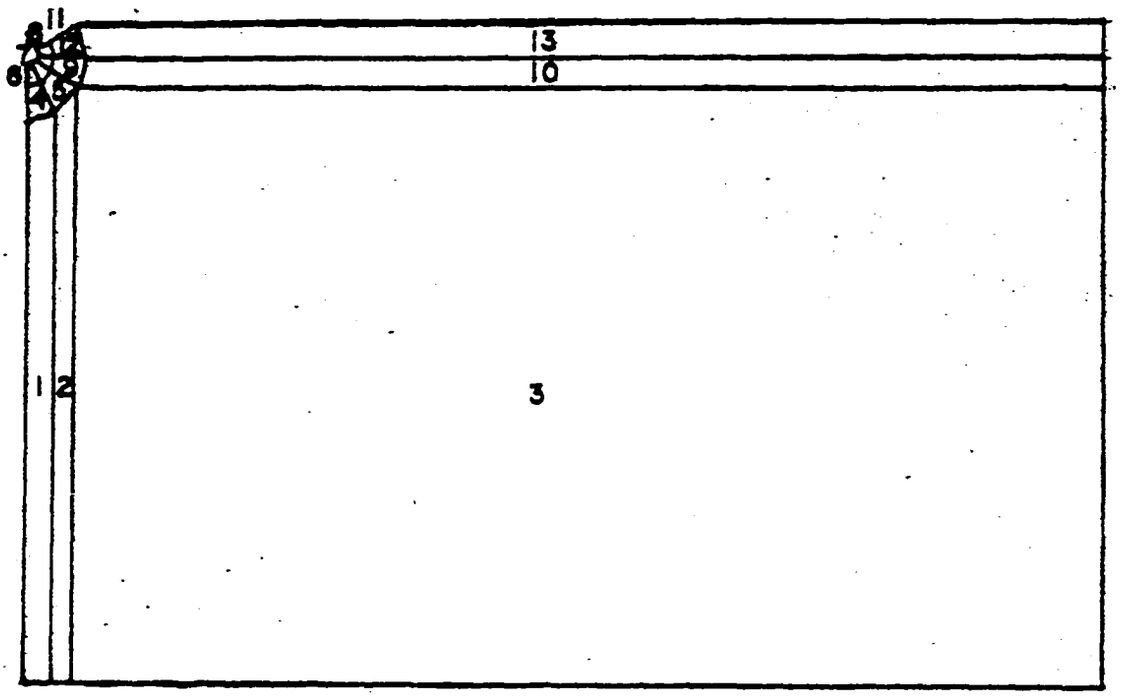


Fig. 6 The initial thirteen regions.

4.1.2 CSubDrainView::OnProjectRunModel()

After initial conditions are defined, this function implements the time marching in the object until a prescribed time is reached, or a convergence criteria is met for the case of steady state. It is assumed that water in the vadose zone above the water table reaches equilibrium instantaneously after the water table elevation decreases.

For each time step, the Laplace equation is solved and the total heads are obtained; then with these total heads the fluxes are calculated using Eq. 25; with these fluxes and recharges a new water table position is obtained. With the average water table position of the past time step and the new water table position, the Laplace equation is solved again, and new fluxes are calculated; using these fluxes and recharges a new water

table position is obtained. Within each time step the process is iterative until two consecutive new water table positions are within a convergence criteria. Then, the time increment is added to the total time and the mesh is redrawn on the screen.

Finally, water tables elevations are calculated with the Kirkham method (Eq. 39), and elevations calculated with the finite element method and Kirkham method are saved in the file wtable.dat.

The functions in the class setup are average_height(), adjust_height(), drain_flux(), calculate_transmissivity(), solve_matrix(), water_table_flux(), and water_table_movement().

4.1.3 CSubDrainView::OnProjectRunRadialModel()

This function is the same as CSubDrainView::OnProjectRunModel() but only the mesh around the drain is drawn.

4.1.4 CSubDrainView::OnProjectRunOneIteration()

This function is similar to CSubDrainView::OnProjectRunModel(), but it executes only one time step, and draws the mesh and flow vectors around the drain.

4.1.5 CSubDrainView::OnProjectViewModel()

Using CSubDrainView::OnDraw(), this function draws the entire mesh.

4.1.6 CSubDrainView::OnProjectViewRadialModel()

This function only draws the mesh around the drain.

4.1.7 CSubDrainView::OnResistanceFittingFunction()

This function obtains the fitting parameters of the logarithmic function ($h(x)/R=\Sigma a_i \ln x^{i-1}$, i=1 to 8) that fits the resistance function: water table heights divided by

recharge for steady state conditions. It is supported by the functions nrutil.c, lfit.c, covsrt.c, and gaussj.c (Numerical Recipies).

4.1.8 CSubDrainView::OnProjectContinueAfterFitting()

This function calculates, with Eqs. 36 to 38 and the finite element method, the water table position, fluxes, and discharge for transient conditions, and it saves the information in the files compare_heights.dat, compare_fluxes.dat. and compare_total_discharges.dat.

4.1.9 OnDraw()

This function draws in the document window the tasks commanded by other View functions: the entire mesh, partial mesh, and resistance function coefficients, etc.

4.2 Cass setup

This is the main class that defines the properties of the objects. Some of its more important functions are SetRegsGrid(), Trigrid(), SetIniSubCondition(), find_radial_elements(), average_height(), adjust_height(), drain_flux(), calculate_transmissivity(), solve_matrix(), water_table_flux(), water_table_movement(), and KirkhamSolution().

4.2.1 setup::SetRegsGrid()

This function sets up the grid for the regions; it defines the large elements (regions), node numbers, and node coordinates.

4.2.2 setup::Trigrid()

This function divides the regions into elements, sets up the finer grid, and optimizes node numeration.

4.2.3 setup::SetIniSubCondition()

This function finds and identifies all of the boundary nodes, radial nodes, column nodes, drain elements, as well as puts all nodes in their initial position. In this initial position, the water table is at the surface.

4.2.4 setup::find_radial_elements()

This function finds and identifies radial elements around the drain.

4.2.5 setup::average_height()

For each time step, this function calculates the average water table position between the past time step and that obtained with the last water table position given by water_table_movement(). Also, it calls adjust_height().

4.2.6 setup::adjust_height()

This function adjusts heights of water table nodes and positions of the rest of the nodes.

4.2.7 setup::drain_flux()

Using the drain transfer coefficient (ALPHA_TILE) and Eq. 34, this function calculates the total flow rate into the drain over a unit width.

4.2.8 setup::calculate_transmissivity()

This function obtains the transmissivity of each element.

4.2.9 setup::solve_matrix()

Using the function element_matrix::stiffness_matrix(), this function does the following tasks: (1) constructs with Eq. 13 the global stiffness matrix, (2) puts the drain element boundary conditions (Neumann's conditions) in the matrix equation; for this task it uses Eqs. 16, 17, 18 and 19, (3) incorporates the prescribed boundary conditions (water

table nodes positions) into the system of equations, and (4) solves the system of equations to obtain the total heads at the nodes.

4.2.10 setup::water_table_flux()

This function calculates the numerator in Eq. 25 that represents fluxes for nodes at the water table surface.

4.2.11 setup::water_table_movement()

This function obtains the new water table position by iteration; for each water table node it gets the first approximation to the height with Eq. 25 (taking the total flux as the sum of the recharge and that obtained with water_table_flux()); then with Newton's method and Eqs. 31, 32 and 33 (derived from the Brook-Corey equations), it refines the estimate of the water table height. In essence this is a water mass balance: the volume of water drained in the unsaturated zone should be equal to the sum of the recharge and the flux in the saturated zone.

4.2.12 setup::KirkhamSolution()

This function returns the value given by Eq. 39.

4.3 Other Classes

4.3.1 Class element_matrix

element_matrix::stiffness_matrix(): For a given element it obtains its element stiffness matrix with Eqs. 13, 16, 17 and 18, and its force vector with Eq. 19; when the element is a drain element, it takes into account the boundary conditions in the drain.

element_matrix::flow_output(): It calculates the output flow in a given element.

4.3.2 Class vecti

This class declares and initializes arrays of integers in one dimension; this class also assigns values to the elements of the arrays. The size of the arrays and the memory allocated are according to need; it uses dynamic memory allocation.

4.3.3 Class vectd

This class is similar to vecti but the arrays are of type double.

4.3.4 Class vectf

This class is similar to vecti but the arrays are of type float.

4.3.5 Class matrixi

This class is similar to vecti but the arrays have two dimensions.

4.3.6 Class matrixd

This class is similar to vectd but the arrays have two dimensions.

4.3.7 Class matrixf

This class is similar to vectf but the arrays have two dimensions.

4.3.8 Class d3_matrixi

This class is similar to vecti but the arrays have three dimensions.

CHAPTER 5

MODEL VALIDATION

Numerical models should be compared with analytical models for validation. An analytical solution to the Laplace equation (Kirkham, 1958) was selected as the most appropriate model for comparison with SubDrain.

The Kirkham equation gives the water table height as a function of the physical characteristics of the drain system and the recharge for steady state regime:

$$h = \frac{LR}{K\pi}\left[\ln\frac{\sin(\pi x/L)}{\sin(\pi r/L)} + \sum_{m=1}^{\infty}\frac{1}{m}\left(\cos\frac{m\pi r}{L/2} - \cos\frac{m\pi x}{L/2}\right)\frac{e^{-m\pi d/(L/2)}}{\sinh(m\pi h/(L/2))}\right] \qquad [39]$$

The meanings of the variables are shown in the fig. 7.

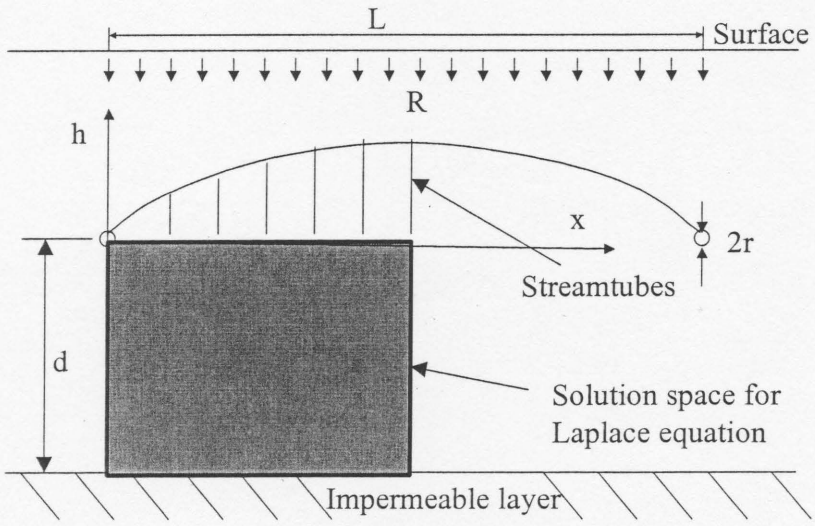


Fig. 7 Kirkham variables for solution to the Laplace equation for subsurface drainage with steady state recharge.

In this section, Kirkham water table profiles are compared with water table profiles calculated with the SubDrain model for steady state recharge with three different soils and six drainage transfer coefficients for each soil.

5.1 Investigated Cases

Three typical soils were investigated (a medium coarse, a medium fine, and a fine) in two-dimensional rectangular drain systems (table 5.1); for these types of soils, the hydraulic conductivity was estimated from Table 16.1 in Smedema and Rycroft (1983). For all simulations, distance from the drain to the impermeable layer, d, was 3 m.

Table 5.1 Soil types, recharge rates, and drain transfer coefficients for comparison of Kirkham analytic solution with SubDrain solution.

Soil	Hyd. Cond. K (m/h)	Drain Spacing L (m)	Recharge R (m/h)	Drainage Transfer Coef. α (h^{-1})
Fine	0.003	60	0.000025	Fixed H on drain boundary
Fine	0.003	60	0.000025	15, 5, 1.5, 0.3, 0.2, and 0.1
Medium fine	0.0085	60	0.00005	Fixed H on drain boundary
Medium fine	0.0085	60	0.00005	10, 5, 1, 0.5, 0.3, and 0.2
Medium coarse	0.05	60	0.0003	Fixed H on drain boundary
Medium coarse	0.05	60	0.0003	40, 10, 5, 3, 2, and 1

The impermeable layer, the horizontal crossing drain center, the vertical middle between drains, and the drain, bound the region solved by Kirkham for two-dimensional flow; also in this analytical solution streamtubes with no resistance to flow were placed

between the drain elevation and the water table surface (fig. 7). Although this geometry is not exactly the same as SubDrain, no analytic solution was found in the literature with a geometry that had a closer approximation to the SubDrain geometry.

5.1.1 Results and Discussion

5.1.1.1 Case 1

For the first scenario (table 5.1) with hydraulic conductivity, K = 0.003 m/h, steady state recharge rate (infiltration) of 0.000025 m/h, and fixed head on the drain surface representing half full drain, the analytic model (Kirkham) water table elevation was greater than the SubDrain water table elevation at the midpoint between drains (fig. 8a). The assumption for the Kirkham analytic solution is that no water flows into the drain above the elevation of the midpoint of the drain. Thus, the analytic solution curve is drawn with the water table intersection with the drain at the drain elevation midpoint. However, the SubDrain model does not restrict flow to the drain to just the lower half of the drain (fig. 8b). Thus, there is less resistance to radial flow into the drain with the SubDrain model than there is with the Kirkham model, and it is logical that far from the drain the SubDrain water table elevation curve is lower than the Kirkham curve, as is shown in fig. 8a. Even though there is not exact agreement between the Kirkham solution and the SubDrain solution, the water table elevation curves from the two models have a similar shape.

The water table elevation curves near the drain for the six drainage transfer coefficients can be seen in fig. 8b. There is a dramatic difference in the water table curves for the different drainage transfer coefficients. The reason for this is that the water table

elevation directly above the drain increases as the drainage transfer coefficient decreases (fig. 8b): as water entrance resistance increases, the head over the drain increases.

The midpoint water table elevation vs. drainage transfer coefficient was plotted in fig. 8c. Although the midpoint water table elevation changes very little, approximately 1% over the range of drainage transfer coefficients, two trends can be observed. For the case where the top of the drain is below the water table, a small change in drainage transfer coefficient results in a relatively high increase in midpoint water table elevation. For the case where the drain is not completely submerged, the midpoint water table elevation is very insensitive to changes in the drainage transfer coefficient, because the wetted perimeter increases as drainage transfer coefficient decreases.

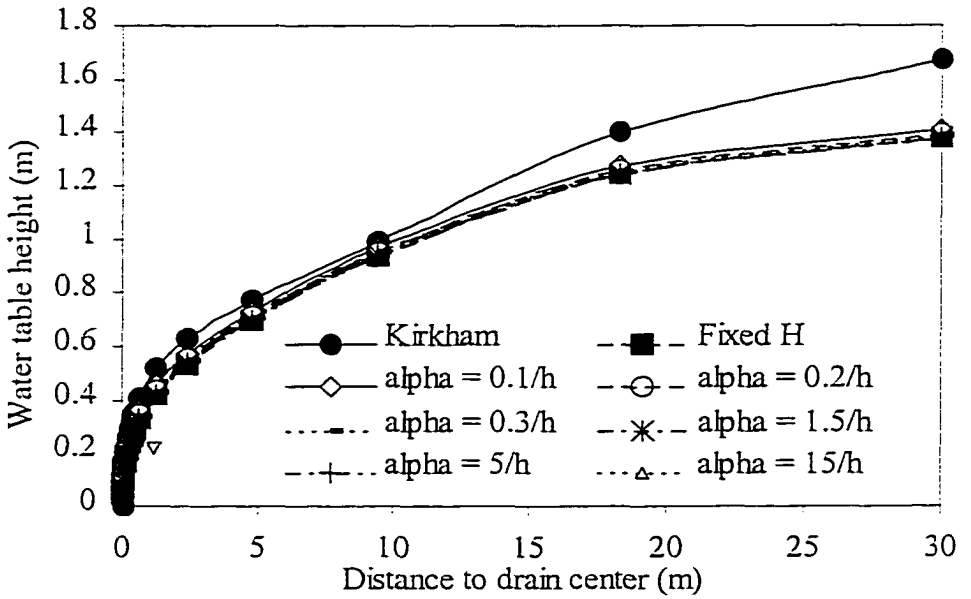


Fig. 8a Kirkham and SubDrain water table elevation vs. distance from the drain: L = 60 m, R = 0.000025 m/h, K = 0.003 m/h, and d = 3 m; alpha = drainage transfer coefficient.

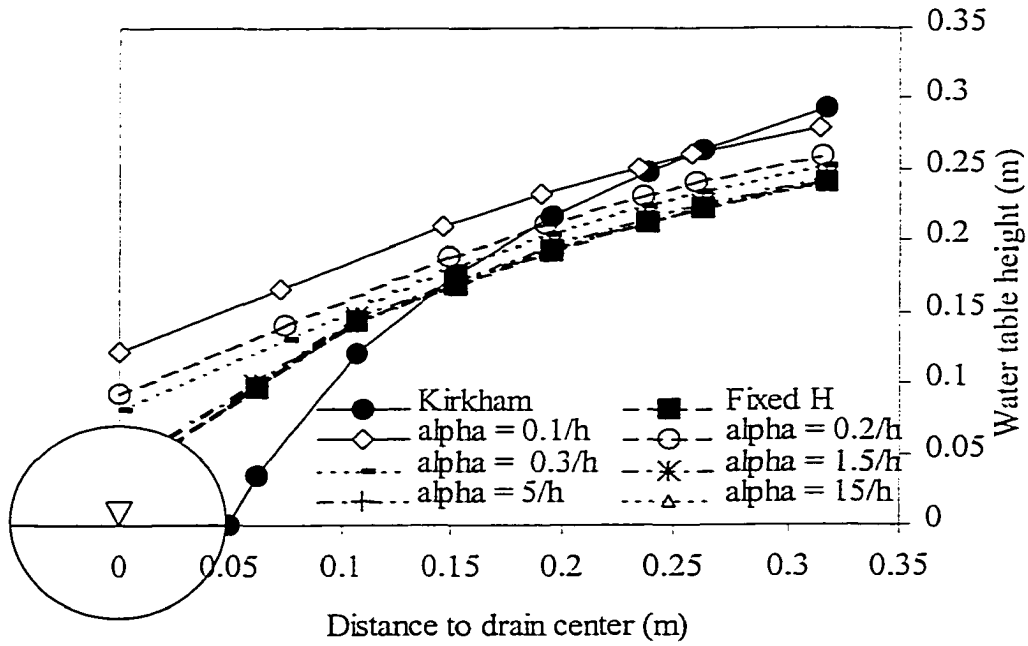


Fig. 8b Kirkham and SubDrain water table elevation vs. distance near the drain: L = 60 m, R = 0.000025 m/h, K = 0.003 m/h, and d = 3 m; alpha = drainage transfer coefficient.

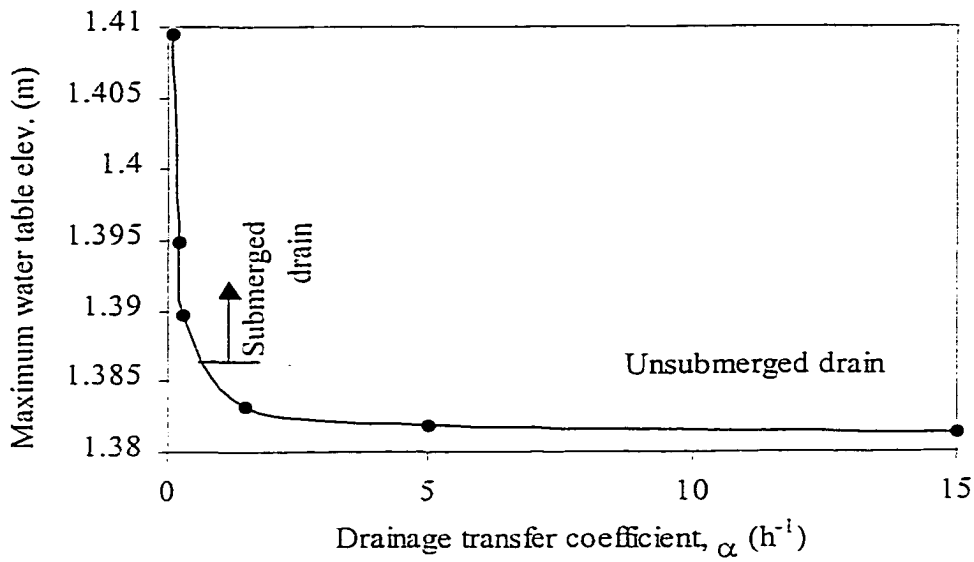


Fig. 8c Water table elevation at midpoint between drains vs. drainage transfer coefficient, alpha: L = 60 m, R = 0.000025 m/h, K = 0.003 m/h, and d = 3 m.

5.1.1.2 Case 2

For the second scenario, a medium fine textured soil (figs. 9a - 9c), the results were very similar to the first scenario; however, the water table profiles for the Kirkham and SubDrain models matched more closely both near the drain (fig. 9b) and up to 2/3 distance to the midpoint between drains (fig. 9a). However, the curves diverged at the midpoint between drains. The ratio between the hydraulic conductivity and the recharge rate was 120 for the first scenario, and was 170 for the second scenario. Thus, the water table was not as high for the second scenario. The curves for the two models appear to match more closely with a lower water table.

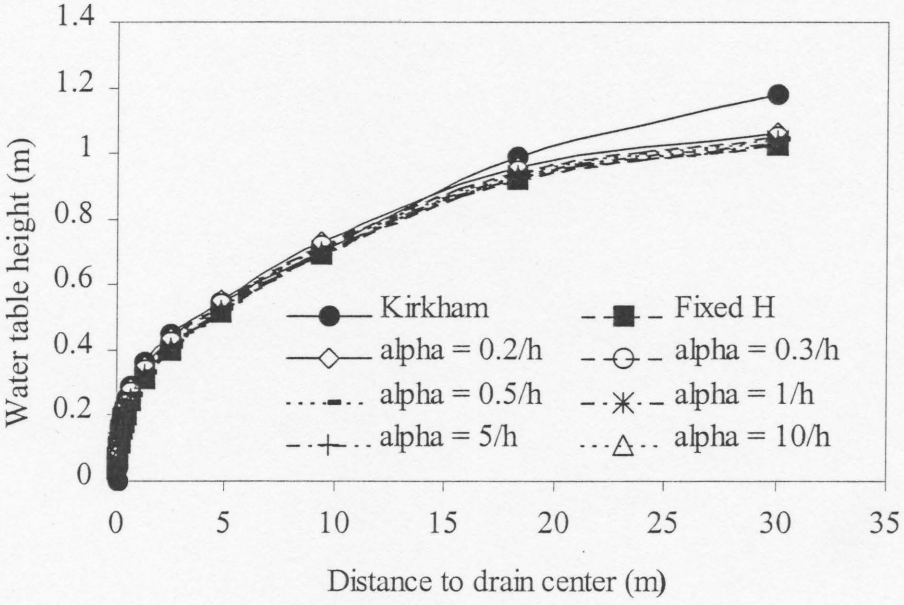


Fig. 9a Kirkham and SubDrain water table elevation vs. distance from the drain: L = 60 m, R = 0.00005 m/h, K = 0.0085 m/h, and d = 3 m; alpha = drainage transfer coefficient.

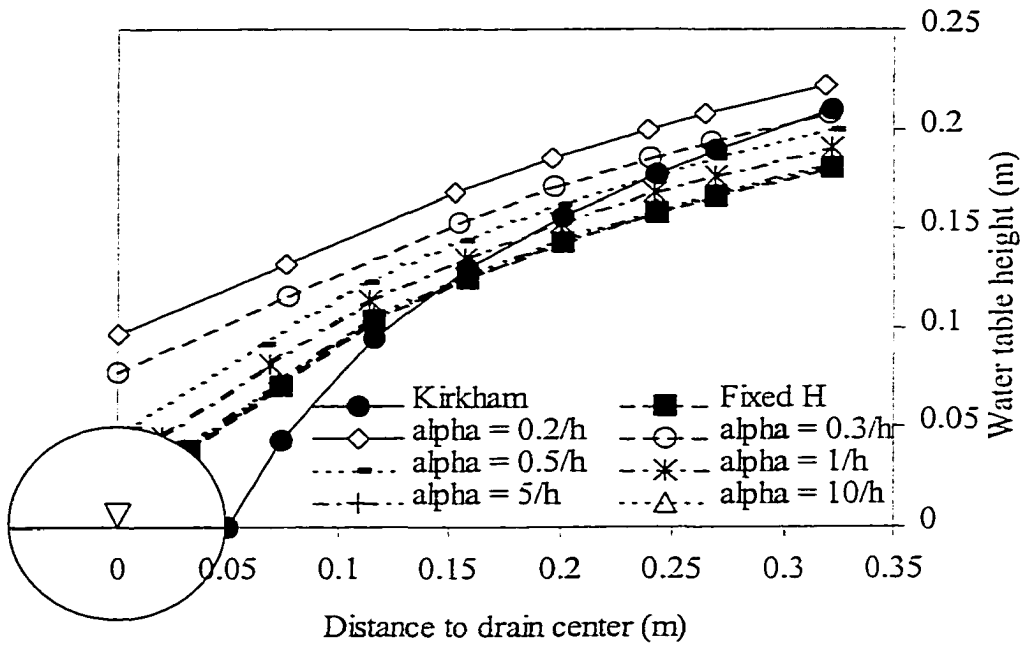


Fig. 9b Kirkham and SubDrain water table elevation vs. distance near the drain: L = 60 m, R = 0.00005 m/h, K = 0.0085 m/h, and d = 3 m; alpha = drainage transfer coefficient.

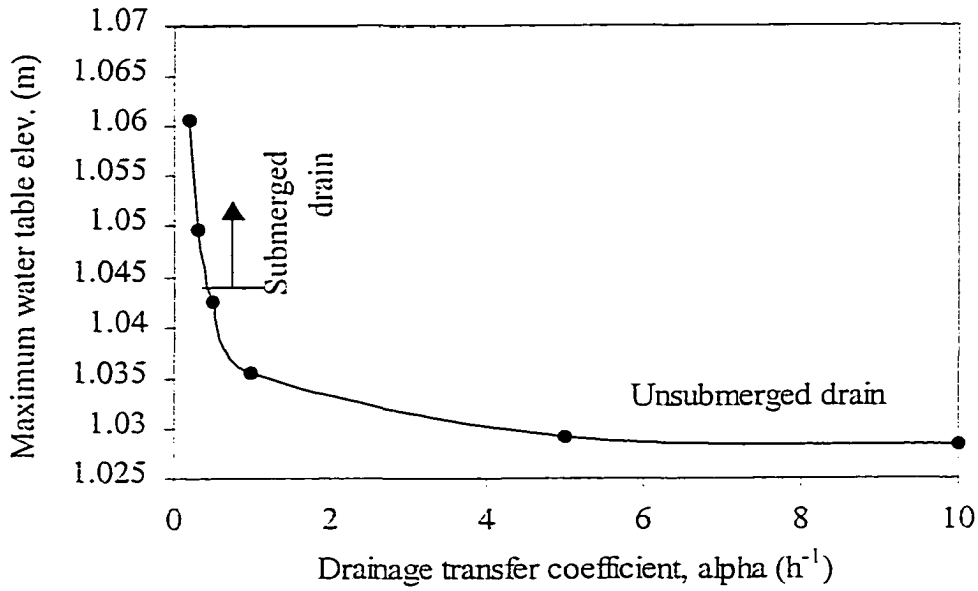


Fig. 9c Water table elevation at midpoint between drains vs. drainage transfer coefficient, alpha: L = 60 m, R = 0.00005 m/h, and K = 0.0085 m/h, and d = 3 m.

5.1.1.3 Case 3

The ratio between the hydraulic conductivity and the recharge rate for scenario 3, the medium coarse textured soil (figs. 10a - 10c), was 167. Thus, the curves for the second and third scenarios were almost identical. The midpoint water table height was relatively insensitive to the drainage transfer coefficient (figs. 10a and 10c). However, midpoint water table height was much more sensitive to the drainage transfer coefficient for the case with the top of the drain submerged below the water table.

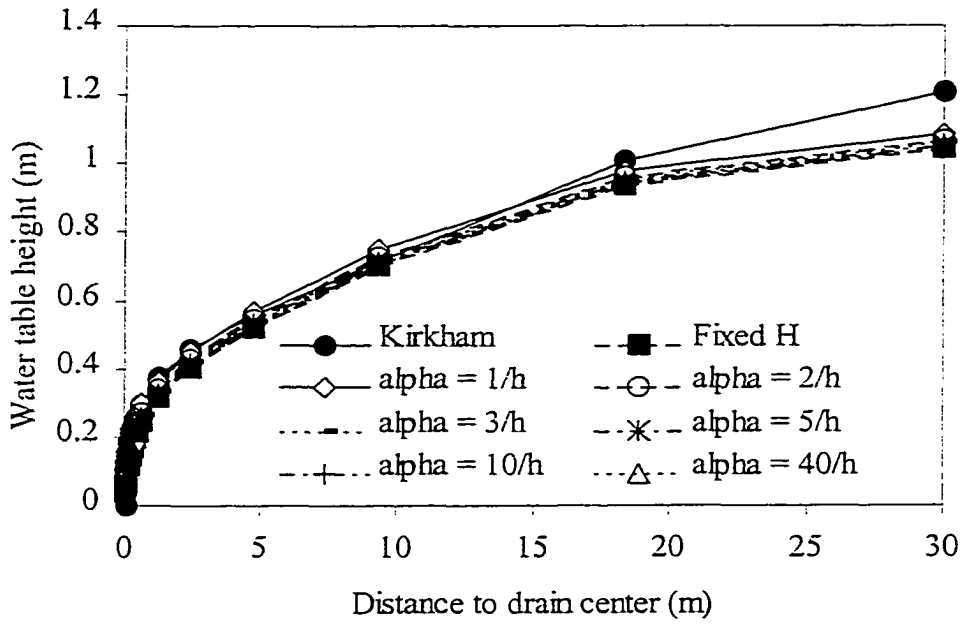


Fig. 10a Kirkham and SubDrain water table elevation vs. distance from the drain: L = 60 m, R = 0.0003 m/h, K = 0.05 m/h, and d = 3 m; alpha = drainage transfer coefficient.

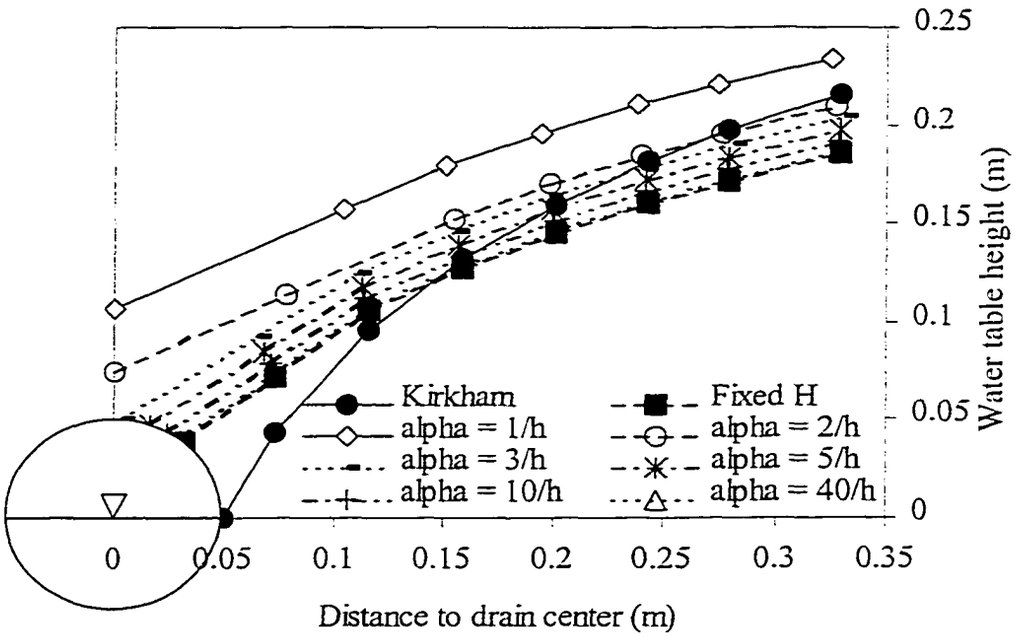


Fig. 10b Kirkham and SubDrain water table elevation vs. distance near the drain: L = 60 m, R = 0.0003 m/h, K = 0.05 m/h, and d = 3 m; alpha = drainage transfer coefficient.

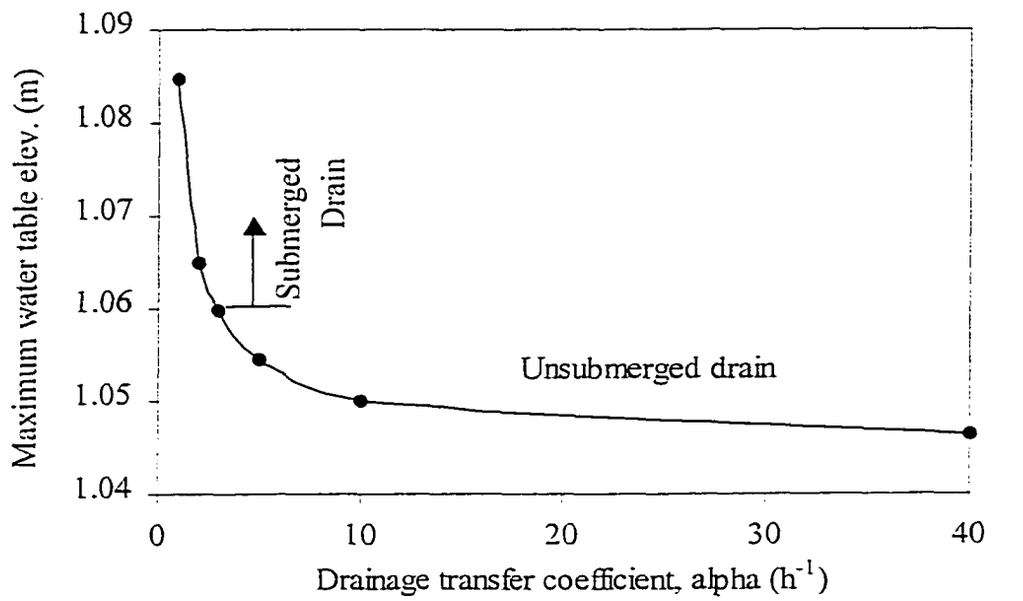


Fig. 10c Water table elevation at midpoint between drains vs. drainage transfer coefficient, alpha: L = 60 m, R = 0.0003 m/h, K = 0.05 m/h, and d = 3 m.

It is likely that the differences between the water table profiles obtained with Kirkham and SubDrain models are due to the physical approximations used by Kirkham to obtain his equation. Since Kirkham neglects seepage in the drain, the wet entry perimeter in the drain with Kirkham theory, u_k, is lesser than that in the numerical model, u_c, and also the maximum water table height in Kirkham, h_{maxK} is greater than that obtained numerically, h_{maxc} (see table 4.2 and fig. 11).

Table 5.2 Comparison of Kirkham and SubDrain maximum water table height; fixed values of potential head in the drain were used.

Soil	h_{maxK} (m)	h_{maxc} (m)	h_{maxK} - h_{max_c} (m)	100 % (h_{maxK}- h_{maxc})/ h_{maxc}	u_K (m)	u_c (m)	u_c-u_K (m)	(h_{maxK}- h_{maxc}) /(u_c-u_K)
Fine	1.67	1.38	0.293	21.2	0.078	0.142	0.063	4.65
Medium Fine	1.18	1.03	0.154	15.0	0.078	0.122	0.043	3.56
Medium coarse	1.21	1.05	0.160	15.3	0.078	0.123	0.044	3.63

h_{maxK} = maximum water table height (midpoint) with Kirkham model, m,
h_{maxc} = maximum water table height (midpoint) with SubDrain model, m,
u_k = wet entry perimeter in Kirkham theory, m,
u_c = wet entry perimeter in the SubDrain model, m.

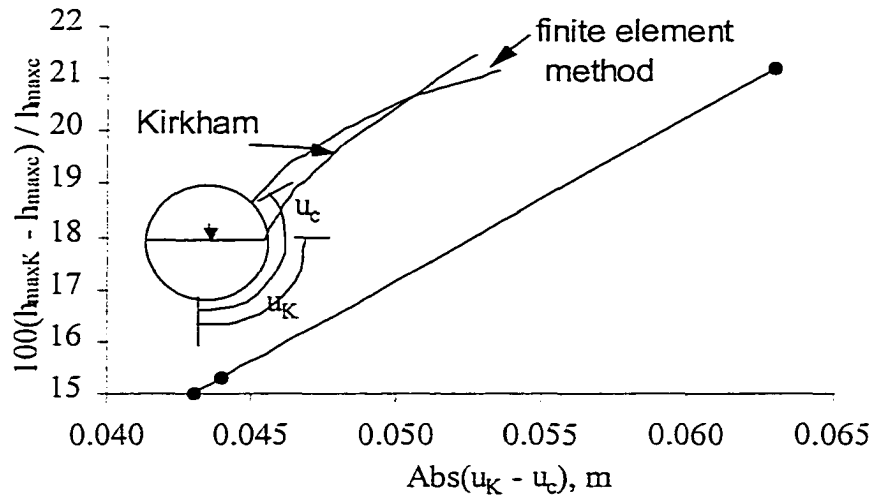


Fig. 11 Relationship between the differences in maximum water table heights and wet entry perimeters for Kirkham and SubDrain method.

The percent difference in maximum water table height is in the range of 15 to 21% between the SubDrain and Kirkham models (fig. 11). Although this would be unacceptable if the wet entry perimeters were the same, it is reasonable considering that water is not allowed to flow into the drain above the midpoint of the drain in the Kirkham model and that water flows toward the drain above its midpoint in the case of the SubDrain program.

For the case where the water table intersects the drain, recharge rate or drain discharge increases with wet entry perimeter, drainage transfer coefficient, and midpoint water table height.

$$Q \propto u\alpha H \qquad [40]$$

where

Q = recharge rate or drain discharge, m^2/h,

u = wet entry perimeter in the drain, m,

α = drain discharge coefficient, h^{-1},

H = difference between midpoint water table height and drain total head, m.

The fact that wet entry perimeter can increase allows the midpoint water table height to be fairly insensitive to drainage transfer coefficient as long as the discharge rate or recharge rate results in an unsubmerged drain. When the drain is not submerged, u increases if α decreases and the change of H is less than it would be if wet entry perimeter were fixed. The insensitivity is clearly shown in figures 8c, 9c, and 10c.

Finally table 5.3 compares the percent difference between the discharge in the drain calculated with drain_flux() and the total recharge at water table level obtained with water_table_flux(). This percent difference is in the range of -11.58 to 14.05 and could indicate there is an error in the drain flux calculation.

Table 5.3 Comparison of drain discharge and total recharge; drain discharge was obtained with the drain_flux() function and total recharge with the water_table_flux() function.

Case	Drainage transfer coefficient, α (h^{-1})	Drain discharge, Q_d ($m^3/h/m$)	Total descharge, Q_r ($m^3/h/m$)	$100(Q_d - Q_r)/Q_r$
1	15	0.000601389	0.000680164	-11.58
2	10	0.00164532	0.00150007	9.68
3	40	0.0102562	0.00899289	14.05

CHAPTER 6

PREDICTING WATER TABLE FALL AND DISCHARGE

Rather than using the finite element model for calculation of water table elevations, fluxes, and discharges it has been proposed to develop semi-analytic equations (equations 36, 37 and 38); these equations require a resistance function which is obtained with the finite element model. The resistance function ($\Omega_f(x) = h(x)/R$) used in the equations is similar to that developed by Kirkham (1964) in his stream tube method, but it can be applied to heterogeneous subdrain systems. The focus of this chapter is on the selection of this function $\Omega_f(x)$.

Following the description of the resistance function, an example is given where water table height and flux for three positions along the water table, and total discharge over time are calculated with the finite element model and equations 37, 38 and 39.

Finally the results of both methods are compared.

6.1 Fitting resistance function

In order to find the best fitted function, regression was used to find the function that had the closest fit to calculated curves, h(x)/R vs. x for steady state rainfall; the calculated curves were generated with the finite element SubDrain model. The Table Curve Program by Jandel Scientific (1993) was used to compare over 1,000 functions with the calculated curves. After extensive analysis, the following equation type was selected as the best fit to the ratio between the water table height and steady state recharge.

$$\Omega_f(x) = \frac{h(x)}{R} = a_1 + \sum_{i=2}^{8} a_i[\log(x)]^{i-1} \qquad [41]$$

A regression program was set up within SubDrain to calculate the coefficients, a_i, i = 1 to 8.

6.2 Example

6.2.1 The problem

With reference to fig. 12 consider the heterogeneous subdrain system with the following characteristics:

Distance between parallel drains, L = 40 m;

Depth from drain center to the second permeable layer, DEPTH1 = 1.0 m;

Depth from drain center to the impermeable layer, DEPTH = 3 m;

Drainage transfer coefficient, $\alpha = 40.0\ h^{-1}$;

Radius of drain tube, RAD = 0.05 m;

Depth of drain below surface, D = 1.5 m;

Rainfall before rain ceases, R = 0.00015 m/h;

Hydraulic conductivity of the first layer, K1 = 0.02 m/h;

Hydraulic conductivity of the second layer, K2 = 0.001 m/h;

Water level in the drain, z = 0.0 m (half full drain);

Bubbling pressure, PB = 0.32 m;

Residual water content, TR = 0.21;

Saturated water content, TS = 0.42;

Lambda of the Brooks and Corey equation, LAM = 0.57.

After steady state is reached with a recharge of 0.00015 m/h, it is required to compare the water table height and flux for three nodes, and the total discharge with time obtained with the finite element method and with the equations that use the resistance function $\Omega_f(x)$.

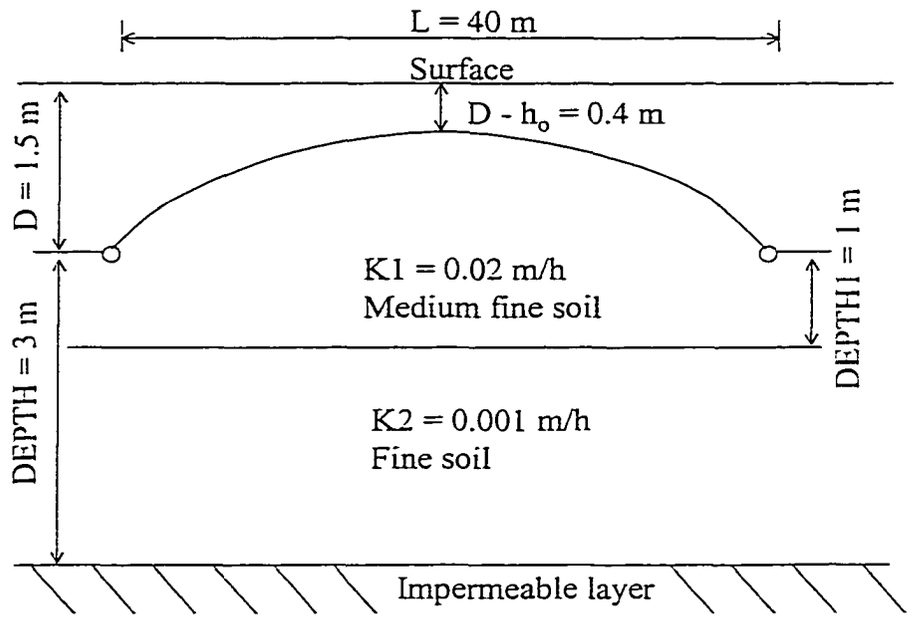


Fig. 12 Drain system characteristics used in the example.

6.2.2 Solution

First, the option Submerged Preprocessor inside the Project item in the menu was run and the grid for the initial condition was obtained (fig. 13); second the option Run Model was executed to obtain the steady state water table height.

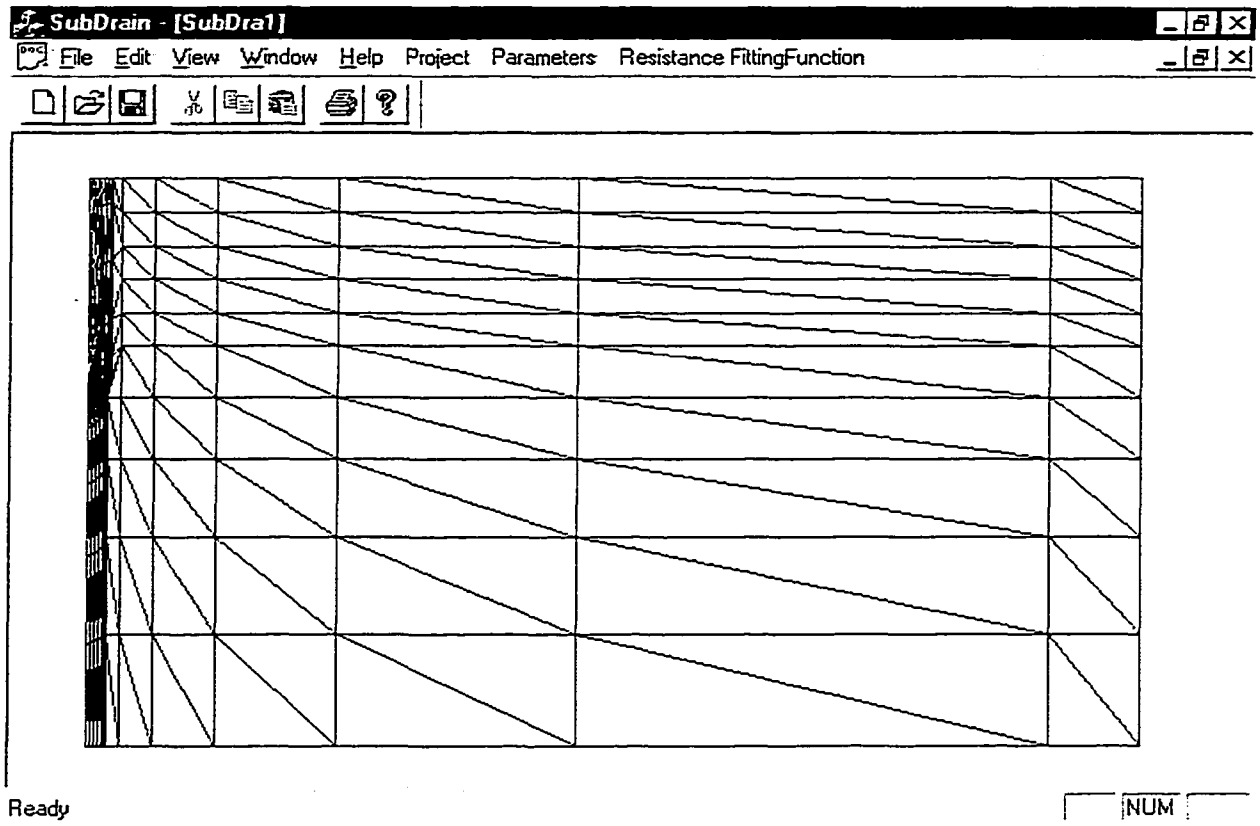


Fig. 13 Grid for the initial condition in the example.

Third, with the Resistance Fitting Function menu option, the regression program within SubDrain gave the following results:

$a_1 = 1913.53,$

$a_2 = 875.803,$

$a_3 = 242.703,$

$a_4 = 87.6469,$

$a_5 = 18.4448,$

$a_6 = -2.25475,$

a_7 = -2.4452,

a_8 = -0.4247

Consequently for the given recharge R = 0.00015 m/h, the resistance function is given by

$$h(x)/R = \Omega_f(x) = 1913.53 + 875.803\log(x) + 242.703[\log(x)]^2 + 87.6469[\log(x)]^3 + 18.4448[\log(x)]^4 - 2.25475\,[\log(x)]^5 - 2.4452[\log(x)]^6 - 0.4247\,[\log(x)]^7 \quad [41]$$

This least squares regression (Eq. 41) and the ratio h(x)/R obtained with the finite element method are shown in fig. 14; both curves are almost identical.

Finally after steady state was reached, the Continue Model After Fitting menu option was run to cease the recharge, R, and to obtain the following results shown in figs. 15-19 and table 6.1.

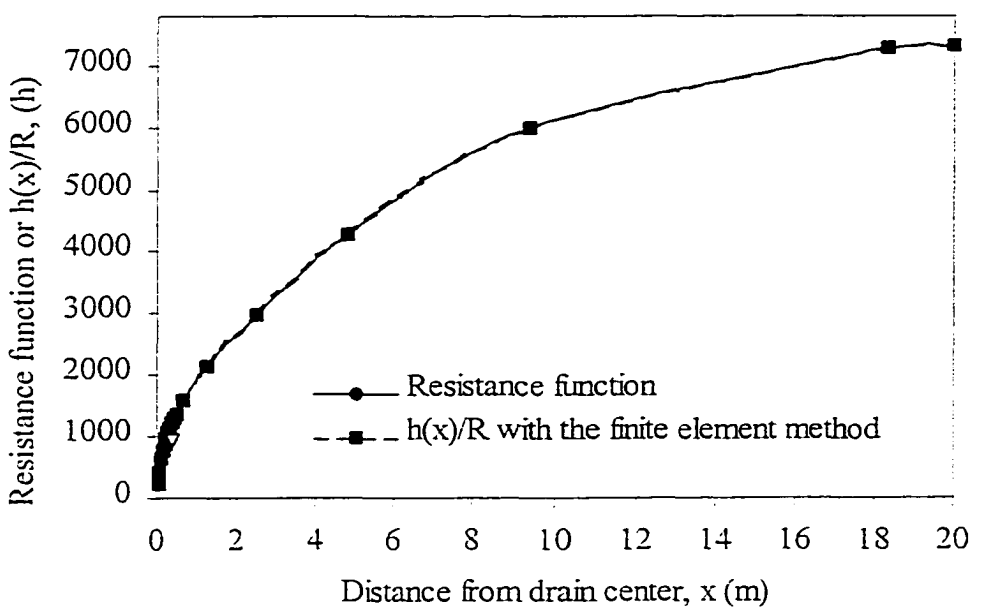


Fig. 14 Resistance function, $\Omega_f(x)$, calculated with the least squares regression and ratio h(x)/R obtained with the finite element method.

Figs. 15 and 16 show the water table positions with time, for the transient regime calculated with equation 36, and with the finite element method. The results are for x = 0.47 m, x = 4.81 m (at 12% of drains distance), and x = 20 m (midpoint between drains).

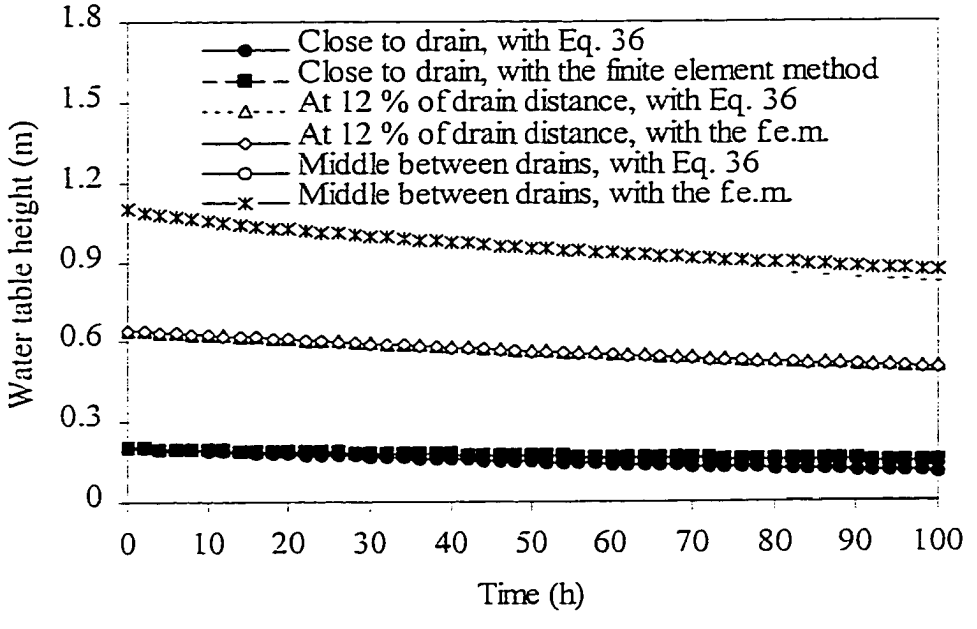


Fig. 15 Water table heights calculated with Eq. 36 and with the finite element method at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x = 4.81 m), and at the middle between drains (x = 20 m).

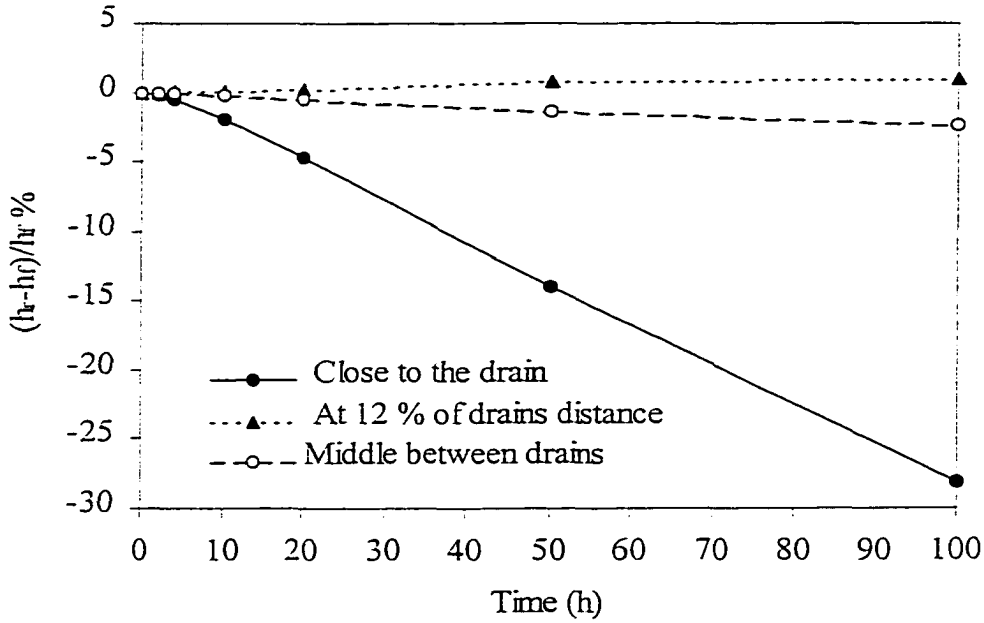


Fig. 16 Water table height differences calculated with Eq. 36 (h_r) and with the finite element method (h_f) at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x = 4.81 m), and at the middle between drains (x = 20 m).

Figs. 17 and 18 give the fluxes at water table for transient regime calculated with bot method.

Fig. 18 shows total discharge per meter of drain vs. time calculated with the two methods.

The difference of total discharge obtained with the two methods is presented in fig. 19.

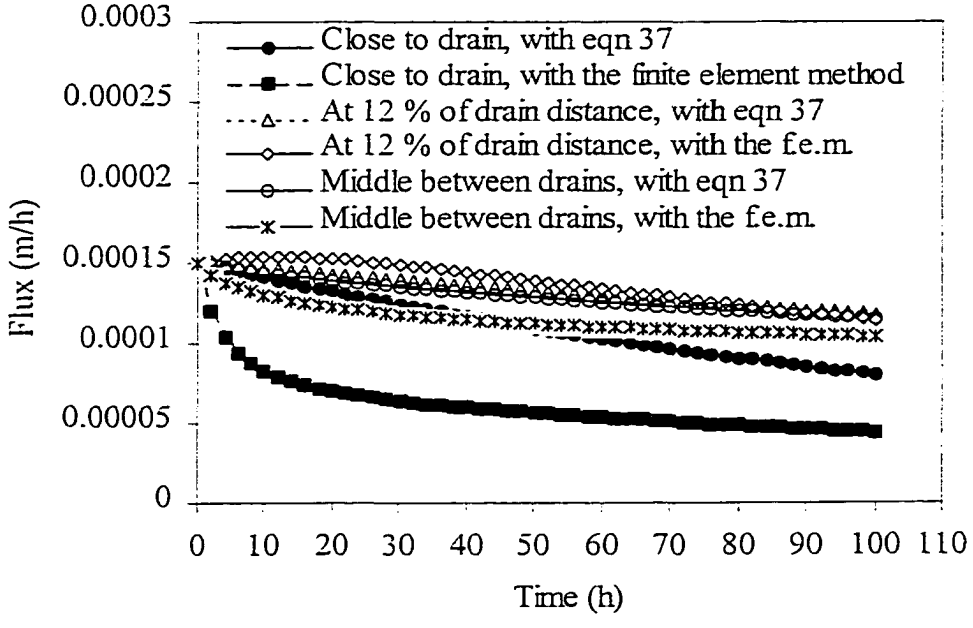


Fig. 17 Fluxes calculated with Eq. 37 and with the finite element method (f_f)at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x = 4.81 m), and at the middle between drains (x = 20 m).

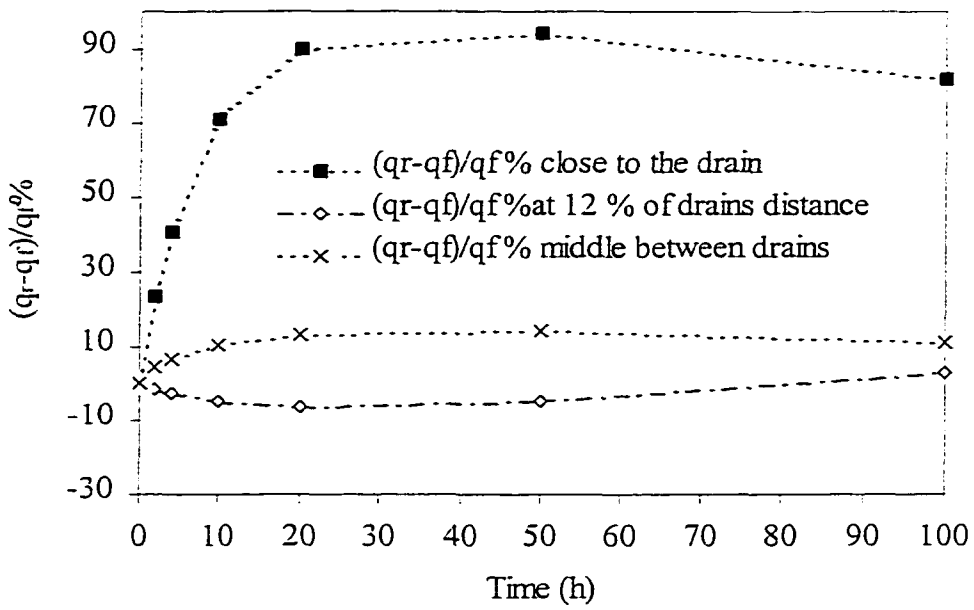


Fig. 18 Flux differences calculated with Eq. 37 (f_r) and with the finite element method (f_f) at three nodes: close to the drain (x = 0.47 m), at 12 % of the drain distance (x=4.81 m), and at the middle between drains (x = 20 m).

Table 6.1 summarizes the results about water table heights and fluxes.

Table 6.1 Comparison of water table heights (m) and fluxes (m/h) calculated with the equations 36 and 37, and with the finite element method at three nodes.

	Time, hours						
	0.0	2.0	4.0	10.0	20.0	50.1	100.1
	Distance to drain, x = 0.47 m (100x/L = 1.2 %)						
h_r	0.20359	0.20087	0.19832	0.19088	0.17912	0.14814	0.10846
h_f	0.20359	0.20116	0.19924	0.19452	0.18804	0.17203	0.15077
$100(h_r - h_f)/ h_f$	0.00	-0.14	-0.46	-1.87	-4.75	-13.89	-28.06
q_r	0.00015	0.00015	0.00015	0.00014	0.00013	0.00011	0.00008
q_f	0.00015	0.00012	0.00010	0.00008	0.00007	0.00006	0.00004
$100(q_r - q_f)/ q_f$	0.00	23.36	40.75	70.73	89.59	94.08	81.85
	Distance to drain, x = 4.81m (100x/L = 12 %)						
h_r	0.64101	0.63753	0.63425	0.62457	0.60892	0.56516	0.50165
h_f	0.64101	0.63750	0.63415	0.62407	0.60746	0.56096	0.49690
$100(h_r - h_f)/ h_f$	0.00	0.01	0.02	0.08	0.24	0.75	0.96
q_r	0.00015	0.00015	0.00015	0.00015	0.00014	0.00013	0.00012
q_f	0.00015	0.00015	0.00015	0.00015	0.00015	0.00014	0.00011
$100(q_r - q_f)/ q_f$	0.00	-1.28	-2.34	-4.59	-6.20	-4.40	3.11
	Distance to drain, x = 20 m (100x/L = 50%)						
h_r	1.09892	1.08723	1.07723	1.05145	1.01656	0.93812	0.84386
h_f	1.09892	1.08751	1.07800	1.05396	1.02208	0.95112	0.86425
$100(h_r - h_f)/ h_f$	0.00	-0.03	-0.07	-0.24	-0.54	-1.37	-2.36
q_r	0.00015	0.00015	0.00015	0.00014	0.00014	0.00013	0.00012
q_f	0.00015	0.00014	0.00014	0.00013	0.00012	0.00011	0.00010
$100(q_r - q_f)/ q_f$	0.00	4.49	6.62	10.20	13.16	14.30	10.89

h_r = water table height predicted using the resistance function.
h_f = water table height calculated with the finite element method.
q_r = water table flux predicted using the resistance function.
q_f = water table flux calculated with the finite element method

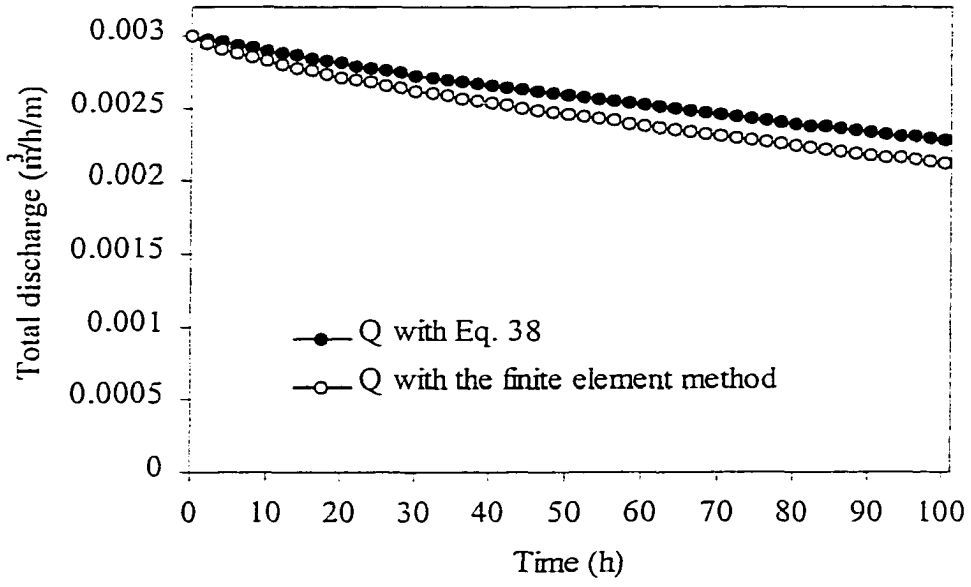


Fig. 19 Total discharge per 1 m of drain vs. time, calculated with Eq. 38 and the finite element method.

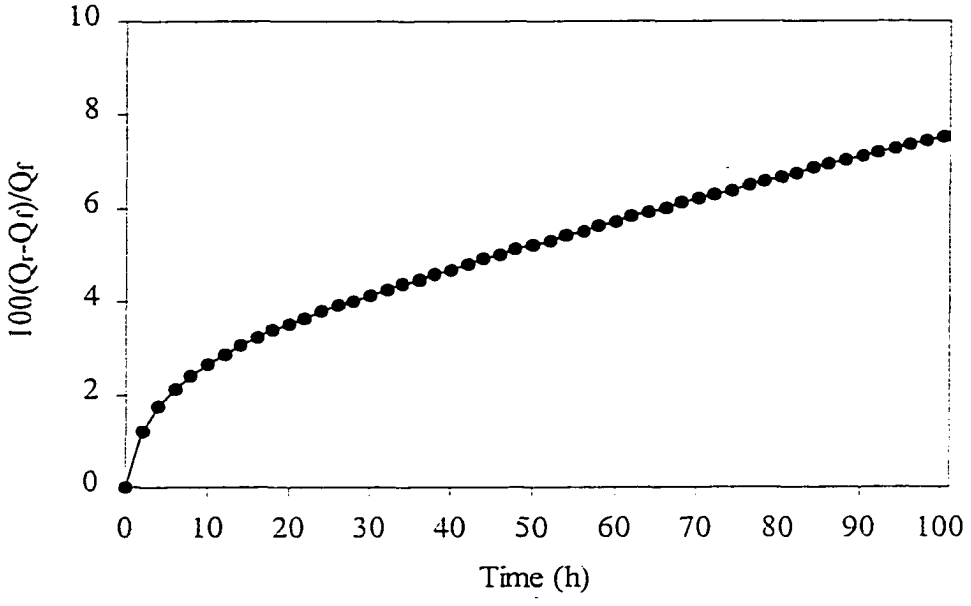


Fig. 20 Percent difference of total discharge obtained with Eq. 38 (Q_r) and the finite element method (Q_f).

6.3 Analysis of Results

According to fig. 14 the resistance function is almost the same as the ratio obtained with the finite element method; hence the resistance function, $\Omega_f(x)$, is a good representation of h(x)/R.

Figs. 15 and 16 show that for the nodes far from the drain (x/L% = 12% and 50 %), the water table elevations calculated with Eq. 36, are close to those obtained with the finite element method. For these nodes the difference between both elevations expressed as $100(h_r - h_f)/ h_f$ ranges between -2.4% to 1%. However for the node near to the drain (x/L% = 1.2%), the agreement is not as good and $100(h_r - h_f)/ h_f$ decreases continuously to reach a minimum of -28%. This fact can be explained if we consider that after the recharge has ceased the wetted perimeter decreases and consequently the resistance to flow close to the drain increases. This increase of resistance is only taken into account by the finite element method, but not by Eq. 36.

Figs. 17 and 18 reveal that at the two nodes far from the drain the agreement of results of both methods (Eq. 37 and finite element method) is satisfactory; the normalized difference, $100(q_r - q_f)/q_f$, ranges from -6.2 to 14.3 during the simulated time. At the node close to the drain, the normalized difference of fluxes with both methods is much greater; $100(q_r - q_f)/q_f$ increases continuously to a maximum value of 94% at 50.1 h after the recharge ceased; between 50.1 and 100.1 h it decreases to 82% at 100.1 h. The explanation of this behavior is the same given for the variation of $100(h_r - h_f)/h_f$ close to the drain: the increase of the resistance to the flow of water is only taken into account by the finite element method.

Even when the fluxes close to the drain obtained with the Eq. 37 and those obtained with the finite element method do not have close agreement, the total discharge values calculated with Eq. 38 and with the finite element method are similar (see figs. 19 and 20). Fig. 19 indicates that the total discharge obtained with Eq. 38, Q_r, is always a little greater than that given by the finite element method, Q_f. The normalized difference of total discharges, expressed as $100(Q_r - Q_f)/Q_f$, and displayed in fig. 20, increases continuously and after 100.1 h reaches a value of only 7.5%.

CHAPTER 7

CONCLUSIONS

- The numerical model SubDrain results are in concordance with the Kirkham equation if the physical approximation (no seepage) of the Kirkham model is taken into account.
- The effect of the fraction of perforated area in the drains can be simulated in the SubDrain model with the drainage transfer coefficient (α).
- Water table elevations calculated with the Eq. 36, h_r, are close to those given by the finite element method, h_f, except near the drain
- Also, fluxes achieved with Eq. 37, q_r, are comparable with those obtained with the finite element method q_f, but not near the drain.
- Discharges calculated with Eq. 38, Q_r, are similar to those obtained with the finite element method, Q_f.
- Eqs. 36 and 37, based in the frozen stream tubes assumption, predicted satisfactory water table elevation and flux for positions not close to the drain.
- Equation 38 (also based in the frozen stream tubes assumption) gave adequate discharges.

APPENDIX A

DRAIN SYSTEM EQUATIONS TO PREDICT DISCHARGE AND WATER TABLE FALL

The purpose of this derivation is to obtain equations that calculate water table height vs. time for any position between drains.

This derivation is similar to that proposed by Kirkham (1964). Referring to fig. A.1, if we assume frozen stream tubes, in an individual stream tube with a node at the water table, the total discharge, Q_{itu}, for steady state recharge is given by Darcy's Law;

$$Q_{itu} = R_s A_{itu} = \left(\frac{\Delta H}{\Omega(x)}\right) A_{itu} \qquad [A.1]$$

where:

A_{itu} = cross sectional area of the tube at the water table, m^2,

R_s = recharge, m/h,

ΔH = total head loss in the stream tube, m,

$\Omega(x)$ = overall stream tube resistance to the flow, h.

From Eq. A.1, we can obtain

$$\frac{\Delta H}{R_s} = \Omega(x) \qquad [A.2]$$

but $\Delta H = h$, then

$$\frac{h}{R_s} = \Omega(x) \qquad [A.3]$$

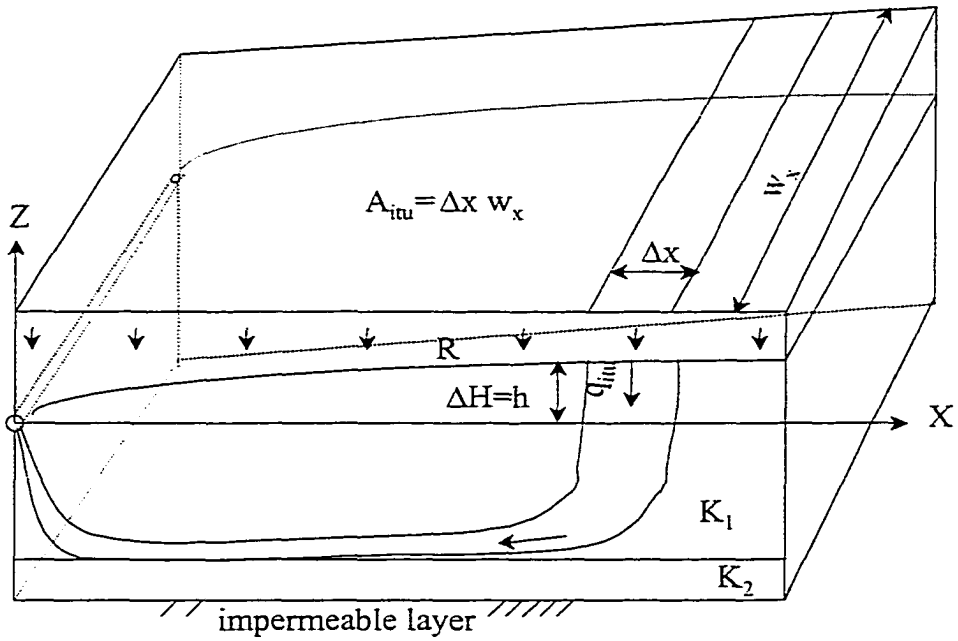


Fig. A.1 Frozen stream tubes and variables of the water table fall equation.

With pairs of values for x and h given by the finite element method (FEM) we can obtain a fitting function $\Omega_f(x)$ equivalent to $\Omega(x)$;

$$\Omega_f(x) = \frac{h}{R_s} \qquad [A.4]$$

For a given water table height at x, the flux in the stream tube at the water table, q_{itu}, can be expressed in two ways:

First for the transient state when R_{tr} is not constant with time

$$Q_{itu} - R_{tr}\Delta x w_x = -f\Delta x w_x \frac{dh}{dt} \qquad [A.5]$$

dividing by $\Delta x w_x$

$$q_{itu} - R_{tr} = -f\frac{dh}{dt} \qquad [A.6]$$

where:

f	=	drainable porosity,
q_{itu}	=	flux in the stream tube at the water table, m/h,
Q_{itu}	=	total discharge in the stream tube, m^3/h,
t	=	time, h,
w_x	=	width of the stream tube at the water table, m,
x	=	x coordinate for midpoint of stream tube at the water table, m,
Δx	=	length of the stream tube at the water table, m .

Second, imagine that the water table (that has fallen at distance x to level h in time t) suddenly stops falling because of an instantaneously recharge, R_s. This R_s, is sufficient to maintain the flow with steady state water table height. R_s would be equal to flux in the tube

$$q_{itu} = R_s \qquad [A.7]$$

Since the tube is frozen the variations of $\Omega_t(x)$ can be neglected and the value of R_s given by equation A.4 can be substituted in equation A.7:

$$q_{itu} = \frac{h}{\Omega_f(x)} \qquad [A.8]$$

From equations A.8 and A.6

$$\frac{h}{\Omega_f(x)} = -f\frac{dh}{dt} + R_{tr} \qquad [A.9]$$

Rearranging

$$\frac{dh}{dt} + \frac{h}{f\Omega_f(x)} = \frac{R_{tr}}{f} \qquad [A.10]$$

The solution to this differential equation is

$$he^{t/f\Omega_f(x)} = \int \frac{R_{tr}}{f} e^{t/f\Omega_f(x)} dt + C$$

If R_{av} is the average of R_{tr} between t_0 and t_1

$$he^{t/f\Omega_f(x)} = \frac{R_{av}}{f} \int e^{t/f\Omega_f(x)} + C$$

Integrating

$$he^{t/f\Omega_f(x)} = R_{av}\Omega_f(x) e^{t/f\Omega_f(x)} + C$$

Rearranging

$$h = R_{av}\Omega_f(x) + Ce^{-t/f\Omega_f(x)} \qquad [A.11]$$

If $t = t_0$ and $h = h_0$

$$h^0 = R_{av}\Omega_f(x) + Ce^{-t_0/f\Omega_f(x)}$$

$$C = \left(h_0 - R_{av}\Omega_f(x)\right) e^{t_0/f\Omega_f(x)}$$

Substituting C in A.11

$$h = R_{av}\Omega_f(x) + \left(h^o - R_{av}\Omega_f(x)\right) e^{t_0/f\Omega_f(x)} e^{-t/f\Omega_f(x)}$$

If $t = t_1$ ant $\Delta t = t_1 - t_0$

$$h^1 = R_{av}\Omega_f(x) + \left(h^0 - R_{av}\Omega_f(x)\right) e^{-\Delta t/f\Omega_f(x)}$$

Rearranging

$$h^1 = h^0 e^{-\Delta t/f\Omega_f(x)} + R_{av}\Omega_f(x)\left(1 - e^{-\Delta t/f\Omega_f(x)}\right) \qquad [A.12]$$

The flux in x at time t_1 is given by Eq. A.8

$$q_{itu}^{1} = \frac{h^{1}}{\Omega_{f}(x)} \qquad [A.13]$$

Substituting in Eq. A.13 the value of h_1 given by Eq. A.12

$$q_{itu}^{1} = \frac{h^{0} e^{-\Delta t / f\Omega(x)}}{\Omega(x)} + R_{av}\left(1 - e^{-\Delta t / f\Omega(x)}\right)$$

or

$$q_{itu}^{1} = q_{itu}^{0} e^{-\Delta t / f\Omega(x)} + R_{av}\left(1 - e^{-\Delta t / f\Omega(x)}\right) \qquad [A.14]$$

The total discharge for the entire drain shed is the sum of the product of q_{itu}^{1} times Δx_i for each of the n stream tubes.

$$Q_1 = \sum_{i=1}^{n} q_{itu}^{1} \Delta x_i \qquad [A.15]$$

APPENDIX B

CODE: MAIN FUNCTIONS

```
void CSubDrainView::OnProjectSubmergedPreprocessor()
{
        region =1;
        status =1;
        dd.initial_time_step();
        dd.set_top_width();
        dd.SetRegsGrid(); // Generate grid for 13 regions
        dd.Trigrid // optimizes the matrix for 13 regions
        dd.SetIniSubCondition(); // Find all of the boundary nodes
        dd.find_radial_elements();
        Invalidate();
}

void CSubDrainView::OnProjectRunModel()
{
        region = 1;
        double wf;
        int j = -3;
        ofstream ofile2;
        ofile2.open(Root+"wtable.dat");
        if(!ofile2) cerr << "could not open output file";
        int reference_water_table_node = dd.countt_r() + dd.countlp_r()+
                        dd.rad_rows_r();
        double c;
        while(c  > 0.000001||T1max>=dd.t1_r())
        {
                c = fabs((dd.heightp_r(reference_water_table_node) -
                        dd.height_r(reference_water_table_node))/
                        dd.heightp_r(reference_water_table_node));
                        dd.add_time_step();
                double flux_diff = 1.;
                double total_flux_p = 1.;
                if(j == 6) dd.reset_nodes();
                dd.set_heightp(j);
                new_time_step = 0;
                j = 1;
                while(j < 6 && flux_diff > 0.001)
                {
                        j++;
```

```
                    dd.average_height();// This gives ave height
                    dd.drain_flux();
                    new_time_step ++;
                    dd.calculate_transmissivity();
                    dd.solve_matrix();
                    wf = dd.water_table_flux();   // water table flux computed at
                        //average height
                    flux_diff = fabs((wf - total_flux_p) / (wf * 0.5 +
                        total_flux_p * 0.5));
                    total_flux_p = wf;
                    dd.water_table_movement();  // This gives height new
                }
                if(j< 6)dd.add_time();
                dd.find_radial_flux_vectors();
                end_height = dd.height_r(dd.countt_r()
                    + dd.countlp_r()+dd.rad_rows_r()+dd.countu_r());

                Invalidate();
                SendMessage(WM_PAINT);
        }

        //Add the lines to get the water table height
        ofile2 << "    alpha =   "<< "\t";
        ofile2 <<  ALPHA_TILE   << "\n";
        ofile2 << "    recharge =  "<< "\t";
        ofile2 <<   R  << "\n";
        ofile2 << "    Permeability =  "<< "\t";
        ofile2 <<   K1  << "\n";
        ofile2 << "   x "<< "\t";
        ofile2 <<" KirkhamSolution "<< "\t";
        ofile2 << " prog_height "<< "\n";
        for(int i = dd.countt_r() + dd.countlp_r()+ 1;
                i <= dd.countt_r() + dd.countlp_r()+dd.rad_rows_r()
                    +dd.countu_r(); i++)
        {
                ofile2 << dd.x_r(dd.ibn_r(i))<< "\t";
                ofile2 <<dd.KirkhamSolution(dd.x_r(dd.ibn_r(i)))<< "\t";
                ofile2 << dd.height_r(i)<<"\n";
        }
        ofile2.close();
}

void CSubDrainView::OnResistanceFittingFunction()
{       //this function obtains the fitting parameters of
```

```
// of the logarithm fuction that fit the water
// table position.
region = 4;
double SPREAD = 0.1 ;
NTERM = 8;
int i,*ia;
float chisq,*a,*x,*y,*sig,**covar;
ofstream ofile3;
ofile3.open(Root+"FitCoefResFunc.dat");
if(!ofile3) cerr << "could not open output file";
ofile3 << "    DEPTH =  "<< "\t";
ofile3 << DEPTH   << "\n";
ofile3 << "    DEPTH1 =  "<< "\t";
ofile3 << DEPTH1   << "\n";
ofile3 << "    alpha =   "<< "\t";
ofile3 << ALPHA_TILE   << "\n";
ofile3 << "    Permeability K1 =  "<< "\t";
ofile3 <<   K1  << "\n";
ofile3 << "    Permeability K2 =  "<< "\t";
ofile3 <<   K2  << "\n";
ofile3 << "    recharge =  "<< "\t";
ofile3 <<   R  << "\n";
//Getting the number of nodes to the right of drain
int first=1;
int count=dd.countt_r() + dd.countlp_r()+
        dd.rad_rows_r()+dd.countu_r();

for (int k = dd.countt_r() + dd.countlp_r()+1;
        k <=dd.countt_r() + dd.countlp_r()+dd.rad_rows_r();k++)
{
        if(dd.x_r(dd.ibn_r(k+1))>=0.001&first==1)first=k;
}
int     NPT =count-first+1;
ia=ivector(1,NTERM);
a=vector(1,NTERM);
x=vector(1,NPT);
y=vector(1,NPT);
sig=vector(1,NPT);
covar=matrix(1,NTERM,1,NTERM);

for (i=1;i<=NPT;i++)
{
        x[i]=(float) dd.x_r(dd.ibn_r(dd.countt_r()
                + dd.countlp_r()+dd.rad_rows_r()+dd.countu_r()-NPT+i));
```

```
        funcs(x[i],a,NTERM);
        y[i] =(float) (dd.height_r(dd.countt_r()
                + dd.countlp_r()+dd.rad_rows_r()+dd.countu_r()-NPT+i)/R);
        sig[i]=(float) SPREAD;
}

for (i=1;i<=NTERM;i++) ia[i]=1;
lfit(x,y,sig,NPT,a,ia,NTERM,covar,&chisq,funcs);

for (i=1;i<=NTERM;i++)
{
        aw[i]=a[i];
        ofile3 <<"a["<<i<<"] = "<<aw[i]<<"\n";
}
ofile3 <<"x      Calculated    Calculated"<<  "\n";
ofile3 <<"       with fit. fn.    with f.e.m"<<  "\n";
double xx=0.0,hf=0.0;

for(i = first;
        i <= dd.countt_r() + dd.countlp_r()+dd.rad_rows_r()
                +dd.countu_r(); i++)
{
        xx = dd.x_r(dd.ibn_r(i));
        ofile3 << xx << "\t";
        hf = (aw[1]+aw[2]*pow(log(xx),1)+aw[3]*pow(log(xx),2)
                +aw[4]*pow(log(xx),3)+aw[5]*pow(log(xx),4)
                +aw[6]*pow(log(xx),5)+aw[7]*pow(log(xx),6)
                +aw[8]*pow(log(xx),7))*R;
        ofile3 <<hf << "\t";
        ofile3 << dd.height_r(i)<<"\n";
}

free_matrix(covar,1,NTERM,1,NTERM);
free_vector(sig,1,NPT);
free_vector(y,1,NPT);
free_vector(x,1,NPT);
free_vector(a,1,NTERM);
free_ivector(ia,1,NTERM);

ofile3.close();

#undef NRANSI
Invalidate();
SendMessage(WM_PAINT);
```

```
}

void CSubDrainView::OnProjectContinueAfterFitting()
{
        region = 11;
        R = 0.0;
        double wf;
        int j = -3;
        double count=2.0;
        double initial_time=dd.t1_r();
        double t = dd.t1_r() - initial_time;
        double total_discharge = 0.0;
        double dx = 0.0;
        for(int i = dd.countt_r()+ dd.countlp_r() + 1;
                        i <= dd.countt_r()+dd.countlp_r()+dd.rad_rows_r()+dd.countu_r();
                        i++)
        {
                dd.get_steady_height(i);
                initial_height[i]=dd.steady_height_r(i);
                new_height[i]=initial_height[i];
                dd.get_steady_flux(i);
                initial_flux[i]= dd.steady_flux_r(i);
                new_flux[i]=initial_flux[i];
                if(i == dd.countt_r()+ dd.countlp_r() + 1)
                        {
                                dx=(dd.x_r(dd.ibn_r(i+1))-dd.x_r(dd.ibn_r(i)))/2;
                        }
                        else
                        {
                                if(i==dd.countt_r() + dd.countlp_r()
                                        +dd.rad_rows_r()+dd.countu_r())
                                {
                                        dx=(dd.x_r(dd.ibn_r(i))-dd.x_r(dd.ibn_r(i-1)))/2;
                                }
                                else
                                {
                                        dx=(dd.x_r(dd.ibn_r(i+1))-dd.x_r(dd.ibn_r(i)))/2+
                                                (dd.x_r(dd.ibn_r(i))-dd.x_r(dd.ibn_r(i-1)))/2;
                                }

                        }

                        total_discharge = total_discharge + dx*new_flux[i];
        }
```

```
ofstream ofile3;
ofstream ofile4;
ofstream ofile5;
ofile3.open(Root+"heights.dat");
if(!ofile3) cerr << "could not open output file";
ofile3 << "   DEPTH =  "<< "\t";
ofile3 << DEPTH   << "\n";
ofile3 << "   DEPTH1 =  "<< "\t";
ofile3 << DEPTH1   << "\n";
ofile3 << "   alpha =   "<< "\t";
ofile3 << ALPHA_TILE   << "\n";
ofile3 << "   Permeability K1 =  "<< "\t";
ofile3 <<  K1  << "\n";
ofile3 << "   Permeability K2 =  "<< "\t";
ofile3 <<  K2  << "\n";
ofile3 << "x of node close to the drain =  "<< "\t";
ofile3 << dd.x_r(dd.ibn_r(dd.countt_r() + dd.countlp_r()
                          +dd.rad_rows_r()))   << "\n";
ofile3 << "x of node at the middle between drains =  "<< "\t";
ofile3 << dd.x_r(dd.ibn_r(dd.countt_r() + dd.countlp_r()
                          +dd.rad_rows_r()+dd.countu_r()))   << "\n";
ofile3 << "\n";
ofile3 << "      Close to the drain      Middle between drains";
ofile3 << "\n";
ofile3 << "t hr  " ;
ofile3 <<  "Predicted h  Program's h ";
ofile3 <<  "Predicted h  Program's h"   << "\n";

ofile4.open(Root+"fluxes.dat");
if(!ofile4) cerr << "could not open output file";
ofile4 << "   DEPTH =  "<< "\t";
ofile4 << DEPTH   << "\n";
ofile4 << "   DEPTH1 =  "<< "\t";
ofile4 << DEPTH1   << "\n";
ofile4 << "   alpha =   "<< "\t";
ofile4 << ALPHA_TILE   << "\n";
ofile4 << "   Permeability K1 =  "<< "\t";
ofile4 <<  K1  << "\n";
ofile4 << "   Permeability K2 =  "<< "\t";
ofile4 <<  K2  << "\n";
ofile4 << "x of node close to the drain =  "<< "\t";
ofile4 << dd.x_r(dd.ibn_r(dd.countt_r() + dd.countlp_r()
                          +dd.rad_rows_r()))   << "\n";
```

```
ofile4 << "x of node at the middle between drains =  "<< "\t";
ofile4 <<  dd.x_r(dd.ibn_r(dd.countt_r() + dd.countlp_r()
                          +dd.rad_rows_r()+dd.countu_r()))   << "\n";
ofile4 << "\n";
ofile4 << "       Close to the drain       Middle between drains";
ofile4 << "\n";
ofile4 << "t hr  " ;
ofile4 <<  "Predicted q  Program's q ";
ofile4 <<  "Predicted q  Program's q"   << "\n";

ofile5.open(Root+"unsteady_total_discharges.dat");
if(!ofile5) cerr << "could not open output file";
ofile5 << "    DEPTH =  "<< "\t";
ofile5 <<  DEPTH   << "\n";
ofile5 << "    DEPTH1 =  "<< "\t";
ofile5 <<  DEPTH1   << "\n";
ofile5 << "    alpha =   "<< "\t";
ofile5 <<  ALPHA_TILE   << "\n";
ofile5 << "    Permeability K1 =  "<< "\t";
ofile5 <<   K1  << "\n";
ofile5 << "    Permeability K2 =  "<< "\t";
ofile5 <<   K2  << "\n";
ofile5 << "\n";
ofile5 << "t hr "  ;
ofile5 <<  "Predicted Q  " ;
ofile5 <<  "water_table_flux Q";
ofile5 <<  " drain_flux Q"   << "\n";

int m = dd.countt_r() + dd.countlp_r()+dd.rad_rows_r();
int n = dd.countt_r() + dd.countlp_r()+dd.rad_rows_r()+4;
int o = dd.countt_r() + dd.countlp_r()+dd.rad_rows_r()+dd.countu_r();
int p = dd.countt_r() + dd.countlp_r()-3;

ofile3 <<t<< "\t"<< "\t";
ofile3 <<new_height[m]<< "\t"<< "\t""\t";
ofile3 << dd.height_r(m)<< "\t"<< "\t";
ofile3 <<new_height[o]<< "\t"<< "\t";
ofile3 << dd.height_r(o)<<"\n";

ofile4 <<t<< "\t";
ofile4 <<-new_flux[m]<< "\t";
ofile4 <<-dd.flux_r(m)<< "\t";
ofile4 <<-new_flux[o]<< "\t";
ofile4 <<-dd.flux_r(o)<<"\n";
```

```
ofile5 <<t<< "\t";
ofile5 << -total_discharge<<"\t"<<"\t";
ofile5 << -dd.water_table_flux()<<"\t"<<"\t";
ofile5 << dd.drain_flux()<<"\n";

while(t<=T2max)
{
        t = dd.t1_r()-initial_time;
        dd.add_time_step();
        double flux_diff = 1.;
        double total_flux_p = 1.;
        if(j == 6) dd.reset_nodes();
        dd.set_heightp(j);
        new_time_step = 0;
        j = 1;
        total_discharge = 0.0;
        while(j < 6 && flux_diff > 0.001)
        {
                j++;
                dd.average_height();// This gives new av. heights (midpoint)
                new_time_step ++;
                dd.calculate_transmissivity();
                dd.solve_matrix();
                dd.drain_flux();
                wf = dd.water_table_flux();// water table flux computed at
                        //average height
                flux_diff = fabs((wf - total_flux_p) / (wf * 0.5 +
                                total_flux_p *0.5));
                total_flux_p = wf;
                dd.water_table_movement();  // This gives new heights
        }

        if(j< 6)dd.add_time();

        for(i = dd.countt_r()+ dd.countlp_r() + 1;
                i <= dd.countt_r() +dd.countlp_r()+dd.rad_rows_r()
                        +dd.countu_r(); i++)
        {
                fo[i]=(TS-TR)*(1.0-pow((PB/(W-initial_height[i])),LAM));
                double xx = dd.x_r(dd.ibn_r(i));
                omega[i] = aw[1]+aw[2]*pow(log(xx),1)+aw[3]*pow(log(xx),2)
                                +aw[4]*pow(log(xx),3)+aw[5]*pow(log(xx),4)
                                +aw[6]*pow(log(xx),5)+aw[7]*pow(log(xx),6)
```

```
                        +aw[8]*pow(log(xx),7);

        new_height[i]=initial_height[i]
                *exp(-dd.dt_r()/(fo[i]*omega[i]))
                + R*omega[i]*(1-exp(-dd.dt_r()/(fo[i]*omega[i])));

        new_flux[i]=initial_flux[i]
                *exp(-dd.dt_r()/(fo[i]*omega[i]));

        if(i == dd.countt_r()+ dd.countlp_r() + 1)
        {
                dx=(dd.x_r(dd.ibn_r(i+1))-dd.x_r(dd.ibn_r(i)))/2.0;
        }
        else
        {
                if(i==dd.countt_r() + dd.countlp_r()
                        +dd.rad_rows_r()+dd.countu_r())
                {
                        dx=(dd.x_r(dd.ibn_r(i))-dd.x_r(dd.ibn_r(i-1)))/2.0;
                }
                else
                {
                        dx=(dd.x_r(dd.ibn_r(i+1))-dd.x_r(dd.ibn_r(i)))/2.0+
                                (dd.x_r(dd.ibn_r(i)) -
                                dd.x_r(dd.ibn_r(i-1)))/2.0;
                }

        }

        total_discharge = total_discharge + dx*new_flux[i];
}
if(t>=count)
{
        ofile3 <<t<< "\t"<< "\t";
        ofile3 <<new_height[m]<< "\t"<< "\t""\t";
        ofile3 << dd.height_r(m)<< "\t";
        ofile3 <<new_height[o]<< "\t"<< "\t";
        ofile3 << dd.height_r(o)<< "\n";

        ofile4 <<t<< "\t";
        ofile4 <<-new_flux[m]<< "\t";
        ofile4 << -dd.flux_r(m)<< "\t";
        ofile4 <<-new_flux[o]<< "\t";
        ofile4 << -dd.flux_r(o)<<"\n";
```

```
                ofile5 <<t<< "\t";
                ofile5 << -total_discharge<<"\t"<<"\t";
                ofile5 << -dd.water_table_flux()<<"\t"<<"\t";
                ofile5 << dd.drain_flux()<<"\n";

                count=count+2.0;
            }

            for(i = dd.countt_r()+ dd.countlp_r() + 1;
                i <= dd.countt_r() + dd.countlp_r()+
                        dd.rad_rows_r()+dd.countu_r(); i++)
            {
                initial_height[i]=new_height[i];
                initial_flux[i]=new_flux[i];

            }

            Invalidate();
            SendMessage(WM_PAINT);
        }
        ofile3.close();
        ofile4.close();
        ofile5.close();
}

void setup::average_height()
{
        for(int i = 1; i <= countt + countlp +2*radius_rows-1+ countu; i++)
        height.ed(i) = (height.ed(i) + height_p.ed(i)) / 2.;
        adjust_height();
}

void setup::adjust_height(void)
{
        // This function adjusts heights of water table nodes and potitions
        // of the rest of the nodes

        // Calculate upper  placement of water table
        int J = countt + countlp + 1;
        vectd x_o(2*radius_rows-1 + columns + 3), z_o(2*radius_rows-1 + columns + 3);

        //Getting the first left node on the water table before adjusting
        int first_before_adjusting=1;
```

```
for (int i = J; i <=countt+countlp+radius_rows;i++)
{
        while (first_before_adjusting==1 && x.ed(ibn.ei(i+1))>=0.001)
                first_before_adjusting = i;
}

//Getting the coordenates of the water tables nodes
// in the radial region and in two nodes outside this region
// before adjusting

for(int k = 1; k <=J + 2*radius_rows +1 - first_before_adjusting; k++)
{
        x_o.ed(k) = x.ed(ibn.ei(k + first_before_adjusting -1));
        z_o.ed(k) = height.ed( k + first_before_adjusting -1);
}//end of calculation of upper placement of water table

//Getting the polynomial coefficient on the left part
// of the water table

polynomial_coefficients(x_o.ed(1),x_o.ed(2),x_o.ed(3),
                z_o.ed(1),z_o.ed(2),z_o.ed(3));
double ho =cof.ed(1);//heigth of water table in the left vertical

//Putting the nodes on their original position
if(past_firstonvert >= J)
{
        z.ed(rad.ei((past_firstonvert)-J+1,J)) = past_z1;
        x.ed(rad.ei((past_firstonvert)-J+1, J)) = past_x1;
        z.ed(rad.ei((past_firstonvert)-J+1, J-1)) = past_z2;
        x.ed(rad.ei((past_firstonvert)-J+1,J-1)) = past_x2;
        z.ed(rad.ei((past_firstonvert)-J+1, J-2)) = past_z3;
        x.ed(rad.ei((past_firstonvert)-J+1, J-2)) = past_x3;
}

// adjusting the nodes that should be moved from the vertical
// to the right and down

if(ho >= RAD)//submerged
{
        if(ho >= RAD + 0.015)
        {
                if (new_time_step <= 1)
                {
                        first_after_adjusting = 1;
```

```
        for (i = J+1; i <=J-1+radius_rows;i++)
        {
                while (first_after_adjusting==1 && (z.ed(ibn.ei(i))+
                        fabs(initial_z.ed(i+1) - z.ed(ibn.ei(i)))/2)
                        >= ho)
                        first_after_adjusting = i;
        }
        if (first_after_adjusting == 1)
                first_after_adjusting=J+radius_rows-1;
}

past_z1 = z.ed(rad.ei(first_after_adjusting-J+1, J));
past_x1 = x.ed(rad.ei(first_after_adjusting-J+1, J));
past_z2 = z.ed(rad.ei(first_after_adjusting-J+1, J-1));
past_x2 = x.ed(rad.ei(first_after_adjusting-J+1, J-1));
past_z3 = z.ed(rad.ei(first_after_adjusting-J+1, J-2));
past_x3 = x.ed(rad.ei(first_after_adjusting-J+1, J-2));

double a=0., b=0., c=0.;
if (past_firstonvert >= first_after_adjusting+1)
{
        vectd x_p_h(4), z_p_h(4);
        for(int k = 1; k <=3; k++)
        {
                x_p_h.ed(k) = x.ed(ibn.ei(k +
                first_before_adjusting-1));
                z_p_h.ed(k) = height_p.ed( k +
                first_before_adjusting -1);
        }

        polynomial_coefficients(x_p_h.ed(1),
        x_p_h.ed(2),x_p_h.ed(3),
        z_p_h.ed(1),z_p_h.ed(2),z_p_h.ed(3));
        height_p.ed(first_after_adjusting) = z_p_h.ed(1);
        a = cof.ed(1);
        b = cof.ed(2);
        c = cof.ed(3);
}

// Calculates polynomial coefficients for the  three upper nodes
polynomial_coefficients(x_o.ed(1),x_o.ed(2),x_o.ed(3),
                z_o.ed(1),z_o.ed(2),z_o.ed(3));

for(i = first_after_adjusting; i <= first_before_adjusting; i++)
```

```
{
        if(z.ed(rad.ei(i-J+1, J)) >= ho)
        {
                row_polynomial_coefficients(
                        x.ed(rad.ei(i-J+1, J)),
                        x.ed(rad.ei(i-J+1,J-1)),
                        x.ed(rad.ei(i-J+1, J-2)),
                        z.ed(rad.ei(i-J+1, J)),
                        z.ed(rad.ei(i-J+1, J-1)),
                        z.ed(rad.ei(i-J+1, J-2)));

                x.ed(rad.ei(i-countt-countlp, J)) =
                        secant(x.ed(rad.ei(i-countt-countlp, J)),
                                cof.ed(1), cof.ed(2), cof.ed(3),
                                row_cof.ed(1), row_cof.ed(2),
                                row_cof.ed(3));
                double xr = x.ed(rad.ei(i-countt-countlp, J));
                z.ed(rad.ei(i-countt-countlp, J))=
                        cof.ed(1) + cof.ed(2)*xr+cof.ed(3)*xr*xr;

                //calculates positions of lower nodes

                //Calculates the total angle
                int row = i-countt-countlp;
                int m = row -1;
                double ze;
                if (i==first_after_adjusting)ze=
                        past_z1;//z.ed(rad.ei(i-countt-countlp, J));
                else
                        ze=z.ed(rad.ei(i-countt-countlp, J));
                double total_angle = newt_raph_angle(
                        ze, m);
                double pid3 = (total_angle)/(countlp+1);
                int k = 0;
                double d = W;
                double rowx[14];

                for (int j=countt+1;j<=countt+countlp;j++)
                {
                        k++;
                        double delpid3 =  k*pid3;
                        x_left_row(row,j,delpid3,m);
                        z_left_row(row,j,delpid3,m,d);
                        rowx[j] = delpid3;
```

```
                        }
                }
        }

        if (past_firstonvert >= first_after_adjusting+1)
        {
                for (i = first_after_adjusting+1; i
                        <=first_before_adjusting;i++)
                {
                        double xr;
                        xr = x.ed(rad.ei(i-J+1, J));
                        height_p.ed(i)= a + b*xr + c*xr*xr;
                        double h = height_p.ed(i);
                }
        }
        x.ed(ibn.ei(first_after_adjusting))=0.0;
        z.ed(ibn.ei(first_after_adjusting))=ho;
        past_firstonvert = first_after_adjusting;
}
else// just submerged
{
        first_after_adjusting = J;

        past_z1 = z.ed(rad.ei(first_after_adjusting-J+1, J));
        past_x1 = x.ed(rad.ei(first_after_adjusting-J+1, J));
        past_z2 = z.ed(rad.ei(first_after_adjusting-J+1, J-1));
        past_x2 = x.ed(rad.ei(first_after_adjusting-J+1, J-1));
        past_z3 = z.ed(rad.ei(first_after_adjusting-J+1, J-2));
        past_x3 = x.ed(rad.ei(first_after_adjusting-J+1, J-2));

        double a=0., b=0., c=0.;

        if (past_firstonvert >= first_after_adjusting+1)
        {
                vectd x_p_h(4), z_p_h(4);
                for(int k = 1; k <=3; k++)
                {
                        x_p_h.ed(k) = x.ed(ibn.ei(k +
                        first_before_adjusting-1));
                        z_p_h.ed(k) = height_p.ed( k +
                        first_before_adjusting -1);
                }
                polynomial_coefficients(x_p_h.ed(1),
                        x_p_h.ed(2),x_p_h.ed(3),
```

```
                z_p_h.ed(1),z_p_h.ed(2),z_p_h.ed(3));
        height_p.ed(first_after_adjusting) = z_p_h.ed(1);
        a = cof.ed(1);
        b = cof.ed(2);
        c = cof.ed(3);
}

// Calculates polynomial coefficients for the  three upper nodes
polynomial_coefficients(x_o.ed(1),x_o.ed(2),x_o.ed(3),
        z_o.ed(1),z_o.ed(2),z_o.ed(3));

for(i = first_after_adjusting; i <= first_before_adjusting; i++)
{
        if(z.ed(rad.ei(i-J+1, J)) >= ho)
        {
                row_polynomial_coefficients(
                        x.ed(rad.ei(i-J+1, J)),
                        x.ed(rad.ei(i-J+1,J-1)),
                        x.ed(rad.ei(i-J+1, J-2)),
                        z.ed(rad.ei(i-J+1, J)),
                        z.ed(rad.ei(i-J+1, J-1)),
                        z.ed(rad.ei(i-J+1, J-2)));

                x.ed(rad.ei(i-countt-countlp, J)) =
                        secant(x.ed(rad.ei(i-countt-countlp, J)),
                                cof.ed(1), cof.ed(2), cof.ed(3),
                                row_cof.ed(1), row_cof.ed(2),
                                row_cof.ed(3));
                double xr = x.ed(rad.ei(i-countt-countlp, J));
                z.ed(rad.ei(i-countt-countlp, J))=
                        cof.ed(1) + cof.ed(2)*xr+cof.ed(3)*xr*xr;


                //calculates positions of lower nodes

                //Calculates the total angle
                int row = i-countt-countlp;
                int m = row -1;
                double ze;
                ze=z.ed(rad.ei(i-countt-countlp, J));
                double total_angle = newt_raph_angle(ze, m);
                double pid3 = (total_angle)/(countlp+1);
                int k = 0;
                double d = W;
```

```
                                double rowx[14];

                                for (int j=countt+1;j<=countt+countlp;j++)
                                {
                                        k++;
                                        double delpid3 =  k*pid3;
                                        x_left_row(row,j,delpid3,m);
                                        z_left_row(row,j,delpid3,m,d);
                                        rowx[j] = delpid3;
                                }
                        }
                }

                if (past_firstonvert >= first_after_adjusting+1)
                {
                        for (i = first_after_adjusting+1;
                        i <=first_before_adjusting;i++)
                        {
                                double xr;
                                xr = x.ed(rad.ei(i-J+1, J));
                                height_p.ed(i)= a + b*xr + c*xr*xr;
                                double h = height_p.ed(i);
                        }
                }
                if(z.ed(rad.ei(1, J)) <= ho)
                {
                x.ed(ibn.ei(first_after_adjusting))=0.0;
                z.ed(ibn.ei(first_after_adjusting))=ho;
                }
                past_firstonvert = first_after_adjusting;
        }
}
else //no submerged
{
        first_after_adjusting = J;

        past_z1 = z.ed(rad.ei(first_after_adjusting-J+1, J));
        past_x1 = x.ed(rad.ei(first_after_adjusting-J+1, J));
        past_z2 = z.ed(rad.ei(first_after_adjusting-J+1, J-1));
        past_x2 = x.ed(rad.ei(first_after_adjusting-J+1, J-1));
        past_z3 = z.ed(rad.ei(first_after_adjusting-J+1, J-2));
        past_x3 = x.ed(rad.ei(first_after_adjusting-J+1, J-2));

        // Calculates polynomial coefficients for the  three upper nodes
```

```
polynomial_coefficients(x_o.ed(1),x_o.ed(2),x_o.ed(3),
                z_o.ed(1),z_o.ed(2),z_o.ed(3));

for(i = first_after_adjusting; i <= first_before_adjusting; i++)
{
        if(z.ed(rad.ei(i-J+1, J)) >= ho)
        {
                row_polynomial_coefficients(
                        x.ed(rad.ei(i-J+1, J)),
                        x.ed(rad.ei(i-J+1,J-1)),
                        x.ed(rad.ei(i-J+1, J-2)),
                        z.ed(rad.ei(i-J+1, J)),
                        z.ed(rad.ei(i-J+1, J-1)),
                        z.ed(rad.ei(i-J+1, J-2)));

                x.ed(rad.ei(i-countt-countlp, J)) =
                        newton_raphson(
                        x.ed(rad.ei(i-countt-countlp, J)),
                        cof.ed(1), cof.ed(2), cof.ed(3),
                        row_cof.ed(1), row_cof.ed(2), row_cof.ed(3));
                double xr = x.ed(rad.ei(i-countt-countlp, J));
                z.ed(rad.ei(i-countt-countlp, J))=
                        cof.ed(1) + cof.ed(2)*xr+cof.ed(3)*xr*xr;0

                //calculates positions of lower nodes

                //Calculates the total angle
                int row = i-countt-countlp;
                int m = row -1;
                double ze;
                ze=z.ed(rad.ei(i-countt-countlp, J));
                double total_angle = newt_raph_angle(
                        ze, m);
                double pid3 = (total_angle)/(countlp+1);
                int k = 0;
                double d = W;
                double rowx[14];

                for (int j=countt+1;j<=countt+countlp;j++)
                {
                        k++;
                        double delpid3 =  k*pid3;
                        x_left_row(row,j,delpid3,m);
                        z_left_row(row,j,delpid3,m,d);
```

```
                                rowx[j] = delpid3;
                        }
                }


        }
        past_firstonvert = first_after_adjusting;
}

// Calculates  nodes position of rows that  are in the left moving region
// but the upper node is not in the left vertical

for( i = 2; i <= J + radius_rows-first_before_adjusting; i++)
{
        //Calcutates polynomial coefficients of water table and rows
        polynomial_coefficients(x_o.ed(i - 1),
                x_o.ed(i),x_o.ed(i + 1),z_o.ed(i - 1),z_o.ed(i),z_o.ed(i + 1));
        row_polynomial_coefficients(x.ed(rad.ei(i-J+first_before_adjusting, J)),
                x.ed(rad.ei(i-J+first_before_adjusting,J-1)),
                x.ed(rad.ei(i-J+first_before_adjusting, J-2)),
                z.ed(rad.ei(i-J+first_before_adjusting, J)),
                z.ed(rad.ei(i-J+first_before_adjusting, J-1)),
                z.ed(rad.ei(i-J+first_before_adjusting, J-2)));
        //Calculates position of the upper row node just in the water table
        int row = i-J+first_before_adjusting;
        x.ed(rad.ei(i-J+first_before_adjusting, J)) =
                secant(x.ed(rad.ei(i-J+first_before_adjusting, J)),
                cof.ed(1), cof.ed(2), cof.ed(3),
                row_cof.ed(1), row_cof.ed(2), row_cof.ed(3));
        double xr = x.ed(rad.ei(i-J+first_before_adjusting, J));
        z.ed(rad.ei(i-J+first_before_adjusting, J))= cof.ed(1) + cof.ed(2)*xr
                +cof.ed(3)*xr*xr;

        //calculates positions of lower nodes

        //Calculates the total angle
        int m = row -1;
        double test8 = z.ed(rad.ei(i-J+first_before_adjusting,J));
        double total_angle = newt_raph_angle(
                z.ed(rad.ei(i-J+first_before_adjusting, J)), m);
        double pid3 = (total_angle)/(countlp+1);
        int k = 0;
        double d = W;
        //Set up the new positions
```

```
        for (int l=countt+1;l<=countt+countlp+1;l++)
        {
                k++;
                double delpid3 =  k*pid3;
                x_left_row(row,l,delpid3,m);
                z_left_row(row,l,delpid3,m,d);
        }
        x.ed(rad.ei(row,countt+1)) =  (x.ed(rad.ei(row,countt))+
                x.ed(rad.ei(row,countt+2)))/2;
        z.ed(rad.ei(row,countt+1)) =  (z.ed(rad.ei(row,countt))+
                z.ed(rad.ei(row,countt+2)))/2;
} // This closes the radius_rows loop

// Calculates  nodes position of rows that  are in the right moving region
int row;
for(i = J +1 + radius_rows-first_before_adjusting;
        i <=J-1+ 2*radius_rows-first_before_adjusting; i++)
{
        row=i-J+first_before_adjusting;
        int test5=5;
        //Calcutates polynomial coefficients of water table and rows
        polynomial_coefficients(x_o.ed(i - 1),x_o.ed(i),x_o.ed(i + 1),
                z_o.ed(i - 1),z_o.ed(i),z_o.ed(i + 1));
        row_polynomial_coefficients(x.ed(rad.ei(row, J)),
                x.ed(rad.ei(row,J-1)),
                x.ed(rad.ei(row, J-2)),
                z.ed(rad.ei(row, J)),
                z.ed(rad.ei(row, J-1)),
                z.ed(rad.ei(row, J-2)));
        double ccof1=cof.ed(1);
        double ccof2=cof.ed(2);
        double ccof3=cof.ed(3);
        double rcof1=row_cof.ed(1);
        double rcof2=row_cof.ed(2);
        double rcof3=row_cof.ed(3);

        //Calculates  the position of upper node  just in the water table
        x.ed(rad.ei(i-J+first_before_adjusting, J)) = newton_raphson(
                x.ed(rad.ei(i-J+first_before_adjusting, J)),
                cof.ed(1), cof.ed(2), cof.ed(3),
                row_cof.ed(1), row_cof.ed(2), row_cof.ed(3));
        double xr = x.ed(rad.ei(i-J+first_before_adjusting, J));
        z.ed(rad.ei(i-J+first_before_adjusting, J))= cof.ed(1) + cof.ed(2)*xr+
                cof.ed(3)*xr*xr;
```

```
        //Calculates positions of lower nodes
        //Calculates the total angle
        int row = i-J+first_before_adjusting;
        int m = row -1;
        double total_angle = newton_raphson2(
                z.ed(rad.ei(i-J+first_before_adjusting, J)), m);
        double pid3 = (total_angle)/(countlp+1);
        int k = 0;
        double d = W;
        for (int J=countt+1;J<=countt+countlp;J++)
        {
                k++;
                double delpid3 =  k*pid3;
                x_right_row( row, J, delpid3, m);
                z_right_row( row, J, delpid3, m, d);
        }
        x.ed(rad.ei(row,countt+1)) =  (x.ed(rad.ei(row,countt))+
                x.ed(rad.ei(row,countt+2)))/2;
        z.ed(rad.ei(row,countt+1)) =  (z.ed(rad.ei(row,countt))+
          z.ed(rad.ei(row,countt+2)))/2;
} // This closes the 2*radius_rows-1 loop

// Adjust elevations of all column nodes outside radius
for( i = countt + countlp + 2*radius_rows ;
        i <= countt+countlp+2*radius_rows-1+countu; i++)
{
        double h = height.ed(i);
        double b = z.ed(rad.ei(2*radius_rows-1, radiuses * 2));
        double v1 = vert_nodes - hor_rows_l;
        for(int j = hor_rows_l + 1; j <= vert_nodes; j++)
        {
                double v2 = j - hor_rows_l;
                z.ed(col_nod.ei(i, j)) = b + v2 / v1 * (h - b);
        }
}

if(first_after_adjusting >=countt+countlp+2)
{
        z.ed(rad.ei(first_after_adjusting-J+1, J-1))=
                (z.ed(rad.ei(first_after_adjusting-J+2, J-1))
                +z.ed(rad.ei(first_after_adjusting-J, J-1)))/2;
        x.ed(rad.ei(first_after_adjusting-J+1, J-1))=
                (x.ed(rad.ei(first_after_adjusting-J+2, J-1))
```

```
                    +x.ed(rad.ei(first_after_adjusting-J, J-1)))/2;
            z.ed(rad.ei(first_after_adjusting-J+1, J-2))=
                    (z.ed(rad.ei(first_after_adjusting-J+2, J-2))
                    +z.ed(rad.ei(first_after_adjusting-J, J-2)))/2;
            x.ed(rad.ei(first_after_adjusting-J+1, J-2))=
                    (x.ed(rad.ei(first_after_adjusting-J+2,J-2))
                    +x.ed(rad.ei(first_after_adjusting-J, J-2)))/2;
    }

    if(first_after_adjusting == countt+countlp+2)
    {
            x.ed(rad.ei(3,J)) =  (x.ed(rad.ei(2,J))+
                    x.ed(rad.ei(4,J)))/2;
            polynomial_coefficients(x_o.ed(1),
            x_o.ed(2),x_o.ed(3),z_o.ed(1),
                    z_o.ed(2),z_o.ed(3));
            z.ed(rad.ei(3, J))= cof.ed(1) + cof.ed(2)*x.ed(rad.ei(3,J))+
                    cof.ed(3)*x.ed(rad.ei(3,J))*x.ed(rad.ei(3,J));
    }

    // Set elevation head for all boundary nodes in water table

    for( i = first_after_adjusting;
            i <= countt + countlp+2*radius_rows-1+countu; i++)
    {
            height.ed(i) = z.ed(ibn.ei(i));
    }

} // This closes the adjust_height function

double setup::drain_flux(void)
{
    double dtotal_flux = 0.;

    // Calculate flux for all nodes
    double dr;
    for(int k = 1; k <= countt+countlp+1; k++)
    {
            if(k == 1)
            {
                    dr=sqrt((x.ed(ibn.ei(k+1))-x.ed(ibn.ei(k)))
                            *(x.ed(ibn.ei(k+1))-x.ed(ibn.ei(k)))
                            +(z.ed(ibn.ei(k+1))-z.ed(ibn.ei(k)))
                            *(z.ed(ibn.ei(k+1))-z.ed(ibn.ei(k))))/2.0;
```

```
                }
                else
                {
                        if(k == countt+countlp+1)
                        {
                                dr=sqrt((x.ed(ibn.ei(k-1))
                                        -x.ed(ibn.ei(k)))
                                        *(x.ed(ibn.ei(k-1))-x.ed(ibn.ei(k)))
                                        +(z.ed(ibn.ei(k-1))-z.ed(ibn.ei(k)))
                                        *(z.ed(ibn.ei(k-1))-z.ed(ibn.ei(k))))/2.0;
                        }
                        else
                        {
                                dr=sqrt((x.ed(ibn.ei(k-1))
                                        -x.ed(ibn.ei(k)))
                                        *(x.ed(ibn.ei(k-1))-x.ed(ibn.ei(k)))
                                        +(z.ed(ibn.ei(k-1))-z.ed(ibn.ei(k)))
                                        *(z.ed(ibn.ei(k-1))-z.ed(ibn.ei(k))))/2.0+
                                        sqrt((x.ed(ibn.ei(k+1))-x.ed(ibn.ei(k)))
                                        *(x.ed(ibn.ei(k+1))-x.ed(ibn.ei(k)))
                                        +(z.ed(ibn.ei(k+1))-z.ed(ibn.ei(k)))
                                        *(z.ed(ibn.ei(k+1))-z.ed(ibn.ei(k))))/2.0;
                        }
                }
                if(z.ed(ibn.ei(k))>=0.0)
                        flux_d.ed(k) = dr*ALPHA_TILE * (phi.ed(ibn.ei(k))
                                                - z.ed(ibn.ei(k)));
                else
                        flux_d.ed(k) = dr *ALPHA_TILE * (phi.ed(ibn.ei(k)));

                dtotal_flux += flux_d.ed(k);
        } // Closes l loop
        return dtotal_flux;
}

void setup::calculate_transmissivity(void)
{
        div_t x1;
        for(int i = 1; i <= nelem; i++)
        {

                if(zelem >= -DEPTH1)
                {
                        telem_x.ed(i) = K1;
```

```
                    telem_z.ed(i) = K1;
                }
                else
                {
                    telem_x.ed(i) = K2;
                    telem_z.ed(i) = K2;
                }
        }
}

void setup::solve_matrix(void)
{
        element_matrix em(4);
        int jgf = nnp;
        int jgsm = jgf + nnp;
        int jend = jgsm + nnp*nbw;
        vectd a(jend +1);
        for (int i = 1; i <= jend; i++)
                a.ed(i) = 0.0;

        // Generation of system of equations

        int num;
        num = countt+countlp;
        rad.ei(1, 0) = 1;

        for(int kk = 1; kk <= nelem; kk++)
        {
                int idbci = 0, idbc2 = 0;
                for(int ii = 1; ii <= num; ii++)
                {
                        if(idbc.ei(1, ii) == kk)
                        {
                                idbci = ii;
                                idbc2 = idbc.ei(2, ii);//local node number with d.b.c.
                        }
                }
                //Element stiffness matrix

                em.stiffness_matrix(idbci, idbc2,
                    x.ed(rad.ei(1, idbci)),
                    x.ed(rad.ei(1, idbci + 1)),
                    z.ed(rad.ei(1, idbci)),
                    z.ed(rad.ei(1, idbci + 1)),
```

```
            x.ed(nod.ei(kk,1)),
            x.ed(nod.ei(kk,2)),
            x.ed(nod.ei(kk,3)),
            z.ed(nod.ei(kk,1)),
            z.ed(nod.ei(kk,2)),
            z.ed(nod.ei(kk,3)),
            telem_x.ed(kk),
            telem_z.ed(kk));

        // Direct stiffness procedure for global stiffness matrix

        for(int i = 1; i <= 3; i++)
        {
                a.ed(jgf+ nod.ei(kk,i)) += em.ef_r(i);
                for(int j = 1; j <= 3; j++)
                {
                        int jj = nod.ei(kk,j) + 1 - nod.ei(kk,i);
                        if(jj > 0)
                        {
                                int j1 = jgsm + (jj - 1) * nnp + nod.ei(kk,i) -
                                        (jj - 1) * (jj - 2) / 2;
                                a.ed(j1) += em.esm_r(i, j);
                        }
                }
        }
} // This closes the loop which generates the system of equations

//Modification and solution of the system  of equations.

// Incorporation of the specified nodal values into the system of
// equationn using the method of deletion of rows and columns

// When fixed values of the total head are used in the drain
/*      for(i = 1; i <= countt +countlp + 1 ; i++)
{
        int k = ibn.ei(i) - 1;
        for(int j = 2; j <= nbw; j++)
        {
                int m = ibn.ei(i) + j - 1;
                if(m <= nnp)
                {
                        int ij = jgsm + (j - 1) * nnp + ibn.ei(i)
                        - (j - 1) * (j - 2) / 2;
                        if(z.ed(ibn.ei(i))<0.0)
```

```
                        {
                                a.ed(jgf + m) += (-a.ed(ij) * 0.0);
                                a.ed(ij) = 0.0;
                        }
                        else
                        {
                                a.ed(jgf + m) += (-a.ed(ij) * z.ed(ibn.ei(i)));
                                a.ed(ij) = 0.0;
                        }
                }
                if(k > 0)
                {
                        int kj = jgsm + (j - 1) * nnp + k - (j - 1) * (j - 2) / 2;
                        if(z.ed(ibn.ei(i))<0.0)
                        {
                                a.ed(jgf + k) += (-a.ed(kj) * 0.0);
                                a.ed(kj) = 0.0;
                        }
                        else
                        {
                                a.ed(jgf + k) += (-a.ed(kj) * z.ed(ibn.ei(i)));
                                a.ed(kj) = 0.0;
                        }
                        k = k - 1;
                }

        }
        if(z.ed(ibn.ei(i))<0.0)
                a.ed(jgf+ibn.ei(i)) = a.ed(jgsm+ibn.ei(i))*0.0;
        else
                a.ed(jgf+ibn.ei(i)) = a.ed(jgsm+ibn.ei(i))*z.ed(ibn.ei(i)) ;
}*/

int first=1;

for (int j = countt+countlp+1; j <=countt+countlp+radius_rows;j++)
{
        while (first==1 && x.ed(ibn.ei(j+1))>=0.001)
        {
                first=j;
                if(first==countt+countlp+1)
                        first=countt+countlp+2;
        }
}
```

```
for(i = first; i <= countt +countlp + 2*radius_rows-1+
        countu ; i++)
{
        int k = ibn.ei(i) - 1;
        for(int j = 2; j <= nbw; j++)
        {
                int m = ibn.ei(i) + j - 1;
                if(m <= nnp)
                {
                        int ij = jgsm + (j - 1) * nnp + ibn.ei(i)
                        - (j - 1) * (j - 2) / 2;
                        a.ed(jgf + m) += (-a.ed(ij) * height.ed(i));
                        a.ed(ij) = 0.0;
                }
                if(k > 0)
                {
                        int kj = jgsm + (j - 1) * nnp + k - (j - 1) * (j - 2) / 2;
                        a.ed(jgf + k) += (-a.ed(kj) * height.ed(i));
                        a.ed(kj) = 0.0;
                        k = k - 1;
                }
        }
        a.ed(jgf + ibn.ei(i)) = a.ed(jgsm + ibn.ei(i)) * height.ed(i);
}

// Following is the tdfdcmp subroutine
// Decomposes the global stiffness matrix [K]
//into an upper triangular matrix

for(i = 1; i <= nnp - 1; i++)
{
        int mj = i + nbw - 1;
        if(mj > nnp)
                mj = nnp;
        int mk = nbw;
        if(nnp - i + 1 < nbw)
                mk = nnp - i + 1;
        int nd = 0;
        for(int j = i + 1; j <= mj; j++)
        {
                mk -= 1;
                nd += 1;
                int nl = nd + 1;
```

```
                int nk, jk, inl, ink, ii;

                for(int k = 1; k <= mk; k++)
                {
                        nk = nd + k;
                        jk = jgsm + (k - 1) * nnp + j - (k - 1) * (k - 2) / 2;
                        inl = jgsm + (nl - 1) * nnp + i - (nl - 1) * (nl - 2) / 2;
                        ink = jgsm + (nk - 1) * nnp + i - (nk - 1) * (nk - 2) / 2;
                        ii = jgsm + i;
                        a.ed(jk) += (-a.ed(inl) * a.ed(ink) / a.ed(ii));
                }
        }
}

// Following is the slvbd or tdfslvbd subroutine
// Decomposition of the global force vector

for(i = 1; i <= nnp - 1; i++)
{
        int mj = i + nbw - 1;
        if(mj > nnp)
                mj = nnp;
        int lll = 1;
        for(int j = i + 1; j <= mj; j++)
        {
                lll += 1;
                int il = jgsm + (lll - 1) * nnp + i - (lll - 1) * (lll - 2) / 2;
                a.ed(jgf + j) += - a.ed(il) * a.ed(jgf + i) / a.ed(jgsm+i);
        }
}

// Backward subtitution for determination of the nodal values

a.ed(nnp) = a.ed(jgf + nnp) / a.ed(jgsm + nnp);

for(int k = 1; k <= nnp - 1; k++)
{
        i = nnp - k;
        int mj = nbw;
        if((i + nbw - 1) > nnp)
                mj = nnp - i + 1;
        double sum = 0.0;
        for(int j = 2; j <= mj; j++)
        {
```

```
                    int n = i + j - 1;
                    int ij = jgsm + (j - 1) * nnp + i - (j - 1) * (j - 2) / 2;
                    sum += a.ed(ij) * a.ed(n);
                }
                a.ed(i) = (a.ed(jgf +i) - sum) / a.ed(jgsm +i);
        }

        // Put calculated nodal values in phi matrix

        for(i = 1; i <= nnp; i++)
                phi.ed(i) = a.ed(i);
}//end of sove_matrix() function

double setup::water_table_flux(void)
{

        double total_flux = 0.; // total flux across the entire water table

        // Calculate flux for nodes at water table surface

        // countt is the number of nodes on the tile drain and non moving region
        // countlp is the number of nodes on the tile drain and moving region
        // countu is the number of nodes along the top outside the moving region

        int first=1;

        for (int i = countt+countlp+1; i <=countt+countlp+radius_rows;i++)
        {
                while (first==1 && x.ed(ibn.ei(i+1))>=0.001){first=i;}
        }

        for(int kk = first; kk <= countt + countlp +2*radius_rows-1 + countu; kk++)
        {
                int node_1 = ibn.ei(kk), // upper node
                node_2 = below_node.ei(kk), // the lower node on the element
                node_3;
                double flux_1 = 0.,
                flux_2 = 0.,
                dx1 = 0.,
                dx2 = 0.,
                c_x1, c_x2, x_dot1, x_dot2, z_dot1, z_dot2, phi_dot1, phi_dot2;

                for(int i = 1; i <= 2; i++)
                {
```

```
// Uses flux at second node to calculate flux at first node.
// ed and ei refer to the vect class
// rad refers to the radial nodes
// rad.ei(a, b) a refers to the distance from the tile
//and b refers to the position
x_dot1 = x.ed(node_1);
z_dot1 = z.ed(node_1);
x_dot2 = x.ed(node_1);
z_dot2 = z.ed(ibn.ei(kk-1))+(z.ed(below_node.ei(kk))-
        z.ed(ibn.ei(kk-1)))*
        (x_dot2-x.ed(ibn.ei(kk-1)))/(x.ed(below_node.ei(kk))-
        x.ed(ibn.ei(kk-1)));
phi_dot1 = phi.ed(node_1);
if (first!= countt+countlp+1)
{
        if(kk==first)
        {
                if(kk!=countt+countlp+2)
                {
                        if(i == 1)
                        {
                                x_dot2 = x.ed(ibn.ei(kk-1));
                                z_dot2 = z.ed(ibn.ei(kk-1));
                                node_3 = ibn.ei(kk+1);
                                phi_dot2 = phi.ed(ibn.ei(kk-1));
                        }
                        else
                        {
                                node_1 = ibn.ei(kk+1);
                                x_dot1 = x.ed(ibn.ei(kk+1));
                                z_dot1=  z.ed(ibn.ei(kk+1));
                                x_dot2 = x.ed(ibn.ei(kk+1));
                                z_dot2 = z.ed(ibn.ei(kk))+
                                (z.ed(below_node.ei(kk+1))
                                -z.ed(ibn.ei(kk)))*
                                (x_dot2x.ed(ibn.ei(kk)))/
                                (x.ed(below_node.ei(kk+1))
                                -x.ed(ibn.ei(kk)));
                                node_3 = ibn.ei(kk);
                                phi_dot2 = phi.ed(ibn.ei(kk))
                                +(phi.ed(below_node.ei(kk+1))
                                -phi.ed(ibn.ei(kk)))*(x_dot2-x.ed(ibn
                                .ei(kk)))
```

```
                    /(x.ed(below_node.ei(kk+1))-
                    x.ed(ibn.ei(kk)));
            }
    }
    else //k == first and k = countt+countlp+2
    {
            if(i == 1)
            {
                    x_dot1= x.ed(ibn.ei(kk))+
                    0.25*(x.ed(ibn.ei(kk+1))-
                    x.ed(ibn.ei(kk)))*
                    (x.ed(ibn.ei(kk+1))-x.ed(ibn.ei(kk)))/
                    (x.ed(ibn.ei(kk+1))-x.ed(ibn.ei(kk)));

                    z_dot1=
                    z.ed(ibn.ei(kk))+(x_dot1-x.ed(ibn.ei(
                    kk)))*
                    (z.ed(ibn.ei(kk+1))-z.ed(ibn.ei(kk)))/
                    (x.ed(ibn.ei(kk+1))-x.ed(ibn.ei(kk)));

                    phi_dot1= phi.ed(ibn.ei(kk))+
                    (x_dot1-x.ed(ibn.ei(kk)))*
                    (phi.ed(ibn.ei(kk+1))
                    phi.ed(ibn.ei(kk)))
                    /(x.ed(ibn.ei(kk+1))-x.ed(ibn.ei(kk)))
                    ;

                    x_dot2 = x_dot1;

                    z_dot2 = z.ed(ibn.ei(kk))+
                    (x_dot2-x.ed(ibn.ei(kk)))*
                    (z.ed(below_node.ei(kk+1))-
                    z.ed(ibn.ei(kk)))/
                    (x.ed(below_node.ei(kk+1))-
                    x.ed(ibn.ei(kk)));

                    phi_dot2 =phi.ed(ibn.ei(kk))+
                    (x_dot2-x.ed(ibn.ei(kk)))*
                    (phi.ed(below_node.ei(kk+1))-
                    phi.ed(ibn.ei(kk)))/
                    (x.ed(below_node.ei(kk+1))-
                    x.ed(ibn.ei(kk)));

                    node_3=ibn.ei(kk+1);
```

```
                    }
                    else
                    {
                            node_1 = ibn.ei(kk+1);
                            x_dot1 = x.ed(ibn.ei(kk+1));
                            z_dot1 = z.ed(ibn.ei(kk+1));
                            phi_dot1 = phi.ed(ibn.ei(kk+1));
                            x_dot2 = x.ed(ibn.ei(kk+1));
                            z_dot2 = z.ed(ibn.ei(kk))+
                            (z.ed(below_node.ei(kk+1))-
                            z.ed(ibn.ei(kk)))*(x_dot2-
                            x.ed(ibn.ei(kk)))
                            /(x.ed(below_node.ei(kk+1))
                            -x.ed(ibn.ei(kk)));

                            phi_dot2 = phi.ed(ibn.ei(kk))
                            +(phi.ed(below_node.ei(kk+1))
                            -phi.ed(ibn.ei(kk)))*
                            (x_dot2-x.ed(ibn.ei(kk)))/
                            (x.ed(below_node.ei(kk+1))-x.ed(ibn
                            .ei(kk)));
                            node_3 = ibn.ei(kk);
                    }
            }
}
else //k!= countt+countlp+1 and k!=first
{
            if(kk!=countt + countlp +
            2*radius_rows-1 + countu)
            {
                    if(i == 1)
                    {
                            node_3=ibn.ei(kk - 1);
                            phi_dot2 = phi.ed(ibn.ei(kk-1))+
                            (phi.ed(below_node.ei(kk))
                            -phi.ed(ibn.ei(kk-1)))*(x_dot2-
                            x.ed(ibn.ei(kk-1)))/
                            (x.ed(below_node.ei(kk))-
                            x.ed(ibn.ei(kk-1)));
                    }
                    else
                    {
                            node_3 = ibn.ei(kk + 1);
                            phi_dot2 = phi.ed(ibn.ei(kk-1))+
```

```
                                        (phi.ed(below_node.ei(kk))
                                        -phi.ed(ibn.ei(kk-1)))*(x_dot2-
                                        x.ed(ibn.ei(kk-1)))/
                                        (x.ed(below_node.ei(kk))-
                                        x.ed(ibn.ei(kk-1)));
                                }
                        }
                        else//k == the last
                        {
                                if(i == 1)
                                {
                                        node_3=ibn.ei(kk - 1);
                                        phi_dot2 = phi.ed(ibn.ei(kk-1))+
                                        (phi.ed(below_node.ei(kk))
                                        -phi.ed(ibn.ei(kk-1)))*
                                        (x_dot2-x.ed(ibn.ei(kk-1)))/
                                        (x.ed(below_node.ei(kk))-
                                        x.ed(ibn.ei(kk-1)));
                                }
                        }
                }
        }
        else // first == countt+countlp+1
        {
                if(kk==countt+countlp+1)
                {
                        if(i == 1)
                        {
                                x_dot1=x.ed(ibn.ei(kk-1));
                                z_dot1= z.ed(ibn.ei(kk))+
                                (z.ed(ibn.ei(kk+1))-z.ed(ibn.ei(kk)))*
                                (x_dot1-x.ed(ibn.ei(kk)))/
                                (x.ed(ibn.ei(kk+1))-x.ed(ibn.ei(kk)));

                                phi_dot1= phi.ed(ibn.ei(kk))+
                                (phi.ed(ibn.ei(kk+1))-phi.ed(ibn.ei(kk)))*
                                (x_dot1-x.ed(ibn.ei(kk)))/
                                (x.ed(ibn.ei(kk+1))
                                -x.ed(ibn.ei(kk)));

                                x_dot2=x.ed(ibn.ei(kk-1));
                                z_dot2= z.ed(ibn.ei(kk))+
                                (z.ed(below_node.ei(kk+1))-
                                z.ed(ibn.ei(kk)))*(x_dot2-x.ed(ibn.ei(kk)))/
```

```
                    (x.ed(below_node.ei(kk+1))-x.ed(ibn.ei(kk))
                    );

                    phi_dot2= phi.ed(ibn.ei(kk))+
                    (phi.ed(below_node.ei(kk+1))-
                    phi.ed(ibn.ei(kk)))*
                    (x_dot2-x.ed(ibn.ei(kk)))/
                    (x.ed(below_node.ei(kk+1))-
                    x.ed(ibn.ei(kk)));

                    node_3 = ibn.ei(kk+1);
            }
            else
            {
                    x_dot2 = x.ed(ibn.ei(kk+1));
                    z_dot2= z.ed(ibn.ei(kk))+
                    (z.ed(below_node.ei(kk+1))-
                    z.ed(ibn.ei(kk)))*(x_dot2-x.ed(ibn.ei(kk)))/
                    (x.ed(below_node.ei(kk+1))-
                    x.ed(ibn.ei(kk)));

                    node_3=ibn.ei(kk+1);
                    phi_dot2= phi.ed(ibn.ei(kk))+
                    (phi.ed(below_node.ei(kk+1))-
                    phi.ed(ibn.ei(kk)))*(x_dot2-
                    x.ed(ibn.ei(kk)))/
                    (x.ed(below_node.ei(kk+1))-
                    x.ed(ibn.ei(kk)));

                    node_3=ibn.ei(kk+1);
            }

    }
    else//k != countt+countlp+1
    {
            if(kk==countt+countlp+2)
            {
                    if(i == 1)
                    {
                            node_3=ibn.ei(kk - 1);
                            phi_dot2 = phi.ed(ibn.ei(kk-1))+
                            (phi.ed(below_node.ei(kk))
                            -phi.ed(ibn.ei(kk-1)))*
                            (x_dot2-x.ed(ibn.ei(kk-1)))/
```

```
                    (x.ed(below_node.ei(kk))-
                    x.ed(ibn.ei(kk-1)));
            }
            else
            {
                    node_3 = ibn.ei(kk + 1);
                    phi_dot2 = phi.ed(ibn.ei(kk-1))+
                    (phi.ed(below_node.ei(kk))
                    -phi.ed(ibn.ei(kk-1)))*
                    (x_dot2-x.ed(ibn.ei(kk-1)))/
                    (x.ed(below_node.ei(kk))-
                    x.ed(ibn.ei(kk-1)));
            }
    }
    else
    {
            if(kk!=countt + countlp +2*radius_rows-1
                    + countu)
            {
                    if(i == 1)
                    {
                            node_3=ibn.ei(kk - 1);
                            phi_dot2 =
                            phi.ed(ibn.ei(kk-1))+
                            (phi.ed(below_node.ei(kk))
                            -phi.ed(ibn.ei(kk-1)))*
                            (x_dot2-x.ed(ibn.ei(kk-1)))/
                            (x.ed(below_node.ei(kk))-
                            x.ed(ibn.ei(kk-1)));
                    }
                    else
                    {
                            node_3 = ibn.ei(kk + 1);
                            phi_dot2 =
                            phi.ed(ibn.ei(kk-1))+
                            (phi.ed(below_node.ei(kk))
                            -phi.ed(ibn.ei(kk-1)))*
                            (x_dot2-x.ed(ibn.ei(kk-1)))/
                            (x.ed(below_node.ei(kk))-
                            x.ed(ibn.ei(kk-1)));
                    }
            }
            else //the last
            {
```

```
                                    if(i == 1)
                                    {
                                            node_3=ibn.ei(kk - 1);
                                            phi_dot2 =
                                            phi.ed(ibn.ei(kk-1))+(phi.ed(
                                            below_node.ei(kk))-phi.ed(ib
                                            n.ei(kk-1)))*(x_dot2-x.ed(ibn
                                            .ei(kk-1)))/(x.ed(below_node.
                                            ei(kk))-x.ed(ibn.ei(kk-1)));
                                    }
                            }
                    }
            }
}

// Centroid for calculation of transmissivity

double em_xelem = (x_dot1+x_dot2+x.ed(node_3)) / 3.;
double em_zelem = (z_dot1+z_dot2+z.ed(node_3)) / 3.;
if(i == 1){c_x1=em_xelem;}
else{c_x2=em_xelem;}

double kz = calculate_transmissivity_node(em_xelem, em_zelem);
double kx = kz;

// This function calculates the flow in the element
element_matrix em(4);
em.flow_output(x_dot1,
x_dot2,
x.ed(node_3),
z_dot1,
z_dot2,
z.ed(node_3),
kx,
kz,
phi_dot1, // potential
phi_dot2,
phi.ed(node_3));
double slope;
slope = (z.ed(node_3) - z_dot1)/(x.ed(node_3) - x_dot1);

// Calculates the flux
double flux = em.gradz_r() - em.gradx_r() * slope;
```

```
if (first!= countt+countlp+1)
{
        if(kk==first)
        {
                if(i == 1)
                {
                        flux_1 = flux;
                        dx1 = 0.25 * fabs(x.ed(ibn.ei(kk))
                        -x.ed(ibn.ei(kk+1)));
                }
                else
                {
                        flux_2 = flux;
                        dx2 = 0.25 * fabs(x.ed(ibn.ei(kk)) -
                        x.ed(ibn.ei(kk+1)));
                }
        }
        else //k!= countt+countlp+1 & k!=first
        {
                if(kk==first+1)
                {
                        if(i == 1)
                        {
                                flux_1 = flux;
                                dx1 = 0.5*fabs(x.ed(ibn.ei(kk)) -
                                x.ed(ibn.ei(kk+1)));
                        }
                        else
                        {
                                flux_2 = flux;
                                dx2 = 0.5*fabs(x.ed(ibn.ei(kk)) -
                                x.ed(ibn.ei(kk-1)));
                        }
                }
                else
                {
                        if(kk!=countt+countlp+2*radius_rows-1 +
                                countu)
                        {
                                if(i == 1)
                                {
                                        flux_1 = flux;
                                        dx1=0.5*fabs(x.ed(ibn.ei(kk)
                                        ) - x.ed(ibn.ei(kk+1)));
```

```
                        }
                        else
                        {
                                flux_2 = flux;
                                dx2=0.5*fabs(x.ed(ibn.ei(kk)
                                ) - x.ed(ibn.ei(kk-1)));
                        }
                }
                else//k == the last
                {
                        if(i == 1)
                        {
                                flux_1 = flux;
                                dx1=0.5*fabs(x.ed(ibn.ei(kk)
                                ) - x.ed(ibn.ei(kk-1)));
                                i++;
                        }

                }
        }
    }
}
else // first == countt+countlp+1
{
        if(kk==countt+countlp+1)
        {
                if(i == 1)
                {
                        flux_1 = flux;
                        dx1 = 0.25 * fabs(x.ed(ibn.ei(kk)) -
                        x.ed(ibn.ei(kk+1)));
                }
                else
                {
                        flux_2 = flux;
                        dx2 = 0.25 * fabs(x.ed(ibn.ei(kk)) -
                        x.ed(ibn.ei(kk+1)));
                }

        }
        else//k != countt+countlp+1
        {
                if(kk==countt+countlp+2)
                {
```

```
                if(i == 1)
                {
                        flux_1 = flux;
                        dx1 = 0.5*fabs(x.ed(ibn.ei(kk)) -
                        x.ed(ibn.ei(kk+1)));
                }
                else
                {
                        flux_2 = flux;
                        dx2 = 0.5*fabs(x.ed(ibn.ei(kk)) -
                        x.ed(ibn.ei(kk-1)));
                }
        }
        else
        {
                 if(kk!=countt + countlp +2*radius_rows-1 +
                        countu)
                {
                        if(i == 1)
                        {
                                flux_1 = flux;
                                dx1=0.5*fabs(x.ed(ibn.ei(kk)
                                ) - x.ed(ibn.ei(kk+1)));
                        }
                        else
                        {
                                flux_2 = flux;
                                dx2=0.5*fabs(x.ed(ibn.ei(kk)
                                ) - x.ed(ibn.ei(kk-1)));
                        }
                }

                else //the last
                {
                        if(i == 1)
                        {
                                flux_1 = flux;
                                dx1=0.5*fabs(x.ed(ibn.ei(kk)
                                ) - x.ed(ibn.ei(kk-1)));
                                i++;
                        }

                }
        }
```

```
                        }
                    }
                } // Closes i loop
                if(kk == countt + countlp + 2*radius_rows-1+countu)
                {
                    flux.ed(kk) = flux_1;
                }
                else
                {
                    flux.ed(kk) =
                    flux_1+(flux_2-flux_1)*(x.ed(ibn.ei(kk))-c_x1)/(c_x2-c_x1);
                }
                total_flux += flux.ed(kk) * (dx1 + dx2);
        } // Closes nodes "k" loop

        return total_flux;
} // Closes water table flux  function

void setup::water_table_movement(void)
{
        vectd vol1(countt + countlp +2*radius_rows+ countu);
        vectd vol1m(countt + countlp +2*radius_rows+ countu);
        double d = W + DEPTH;
        int first=1;

        for (int i = countt+countlp+1; i <=countt+countlp+radius_rows;i++)
        {
            while (first==1 && x.ed(ibn.ei(i+1))>=0.001)first = i;
        }

        for( i = first; i <= countt + countlp + 2*radius_rows-1+ countu ; i++)
        {
            double b = height_p.ed(i) + DEPTH;
            if(height.ed(i) < W - PB)
            {
                vol1.ed(i) = TS * (PB + b) + TR * (d - b - PB) +
                        (TS - TR) * pow(PB, LAM) / (- LAM + 1.)  *
                        (pow(d - b, -LAM + 1.) - pow(PB, -LAM + 1.));
                height.ed(i) = height_p.ed(i) + (R+flux.ed(i))
                        * dt / (TS - TR) / (1.0 - pow((PB / (d - b)), LAM));
            }

            // Compute volume 1 and estimated height for the case:
            // surface water table
```

```
        else
        {
                vol1.ed(i) = TS * d;
                height.ed(i) = W - PB - 0.2;
        }

}

for( i = first; i <= countt + countlp + 2*radius_rows-1+countu ; i++)
{
        if(height.ed(i) > W - PB)
        {
                height.ed(i) = W - PB - 0.2;
        }

        vol1m.ed(i)=vol1.ed(i) +(R + flux.ed(i)) * dt;

        if (vol1m.ed(i)>TS*d)
        {
                vol1m.ed(i)=TS*d;
        }

        double h = height.ed(i) + DEPTH,
        dh = 1.;
        int j = 0;
        while(fabs(dh) > 0.0001)
        {//Newton's method
                double f = - vol1m.ed(i) +
                        TS * (PB + h) + TR * (d - h - PB) +
                        (TS - TR) * pow(PB, LAM) / (- LAM + 1.)
                        * (pow(d - h, -LAM + 1.)
                        - pow(PB, -LAM + 1.));
                double df = (TS - TR) * (1.0 - pow((PB / (d - h)), LAM));
                dh = f / df;
                h -= dh;
                j++;
                if(j > 60)
                {
                        cout << "The water table movement function did not
                                converge\n";
                        exit(0);
                }
        }
```

```
                height.ed(i) = h - DEPTH;
        }
} // Closes the water_table_movement() function
```

REFERENCES

Asseed, M. and D. Kirkham. 1966. Depth of barrier and water table fall in a tile drainage model. Soil Sci. Soc. Am. Proc. 30: 292-298.

Bouwer, H. and J. van Schilfgaarde. 1962. Simplified method of predicting fall of water table in drained land. Transactions of the ASAE. 288-296.

Breve, M.A., R.W. Skaggs, H. Kandil, J.E. Parsons and J.W. Gilliam. 1992. DRAINMOD-N: A nitrogen model for artificially drained soils. Proceedings of the Sixth International Drainage Symposium. ASAE, St. Joseph, Michigan. 327-336.

Bear, J. 1972. Dynamics of Fluids in Porous Media. Dover Publication, Inc. New York.

Childs, E.C. 1971. Drainage of ground water resting on a sloping bed. Water Resources Research. 7(5): 1256-1263.

Dass, P. and H.J. Morel-Seytoux. 1974. Subsurface drainage solutions by Galerkin's method. Journal of the Irrigation and Drainage Division, ASCE. 100(1): 1-15.

Fipps, G., R.W. Skaggs and J.L. Nieber. 1986. Drain as a boundary condition in finite elements. Water Resources Research. 22(11): 1613-1621.

Fipps, G. 1988. Numerical solutions to the Richards equation in two and three dimensions. PhD thesis, North Carolina State University, Raleigh, U.S.A.

Fipps, G. and R.W. Skaggs. 1989. Influence of slope on subsurface drainage of hillsides. Water Resources Research. 25(7): 1717-1726

Guitjens, J. and J.N. Luthin. 1965. Viscous model study of drain spacing on sloping land and comparison with mathematical solution. Water Resources Research. 1(4): 523-530.

Hammand, H.Y. 1963. Depth and spacing of tile drain systems. Transactions of the ASAE, Paper No. 3517. Vol. 128, Part III, PP. 535-560.

Jury, W.A. 1975. Solute travel-time estimates for tile-drained fields: I. Theory. Soil Sci. Soc. Amer. Proc. 39: 1020-1023.

Jandel Scientific. 1993. TableCurve 2D. AISN software.

Kirkham, D. and R.E. Gaskell. 1951. The falling water table in tile and ditch drainage. Soil Sci. Soc. Amer. Proc. 15: 37-42.

Kirkham, Don. 1958. Seepage of steady rainfall through soil into drains. Am. Geo. U. Trans. 39: 892-908.

Kirkham, Don. 1964. Physical artifices and formulas for approximating water table fall in tile drained land. Soil Sci. Soc. Amer. Proc. 28: 585-590.

Luthin, J.N. and J.C. Guitjens. 1967. Transient solutions for drainage of sloping land. Journal of Irrigation and Drainage, Proceedings of ASCE. 93(3): 43-51.

Marsily,.de G. 1986. Quantitative Hydrogeology. Translated by Gunilla de Marsily. Academic Press, Inc. London.

Merva, G.E., H. Murase, N.R. Fausey. 1981. Finite element modeling for depth spacing of drains in soil of varying hydraulic conductivity. ASAE Paper No. 81-2066, St. Joseph, Michigan: ASAE.

Mohammad, F.S. and R.W. Skaggs. 1983. Drain tube opening effects on inflow. Journal of Irrigation and Drainage Engineering, ASCE. 109(4): 393-404.

Moody, W.T. 1966. Nonlinear differential equation of drain spacing. Journal of the Irrigation and Drainage Division, ASCE. 92(2): 1-9.

Pandey, R.S., A.K. Bhattacharya, O.P. Singh and S.K. Gupta. 1992. Drawdown solutions with variable drainable porosity. Journal of the Irrigation and Drainage Engineering, ASCE. 118(3): 382-396.

Ram, S. and H.S. Chauhan. 1987. Drainage of sloping lands with constant replenishment. Journal of Irrigation and Drainage Engineering, ASCE. 113(2): 213-223.

Schilfgaarde, J. van. 1963. Design of tile drainage for falling water table. Journal of the Irrigation and Drainage Division, ASCE. 89(2): 1-11.

Schmid, P. and J. Luthin. 1964. The drainage of sloping lands. J. of Geophysical Research. 69(8): 1525-1529.

Scotter, D.R., L.K. Heng, D.J. Horne and R.E. White. 1990. A simplified analysis of soil water flow to a mole drain. Journal of Soil Science. 41: 189-198.

Segerlind, L.J. 1984. Applied Finite Element Analysis (2^{nd} ed.). John Wiley and Sons. New York.

Šimůnek, J., T. Vogel and M.Th. van Genuchten. 1994. The SWMS_2D code for simulating water flow and solute transport in two-dimensional variably saturated media, Version 1.21. Research Report No. 132, U.S. Salinity Laboratory, USDA, ARS, Riverside, California.

Smedema, L.K. and D.W. Rycroft. 1983. Land Drainage. Cornell University Press. New York.

Towner, G.D. 1975. Drainage of groundwater resting on a sloping bed with uniform rainfall. Water Resources Research. 11(1): 144-147.

van der Ploeg, R.R., M. Marquardt and D. Kirkham. 1997. On the history of the ellipse equation for soil drainage. Soil Sci. Soc. Amer. J. 61: 1604-1606.

van der Ploeg, R.R., R. Horton, and D. Kirkham. 1999. Steady flow to drains and wells. Agronomy 38: 213-264 (R. W. Skaggs and J. van Schilfgaarde, editors) American Society of Agronomy, Madison, Wisconsin.

Waller, P. and D.B. Jaynes. 1993. Modeling tile drainage in an irregular network. ASAE Summer Meeting, Spokane, Washington.

Yu, S.C. and K.D. Konhya. 1992. Boundary modeling of steady saturated flow to drain tubes. Proceedings of the Sixth International Drainage Symposium. ASAE, St. Joseph, Michigan. 305-312.