One-dimensional saturated and unsaturated hydrological simulations: *Applying case studies for verification and checking the performance of the Flow model written in Python*

Bram Berendsen

August 23, 2021

Abstract

This study introduces a hydrological model for one-dimensional saturated and unsaturated flow simulations using the modern programming language Python. This *Flow* model implements a backward Euler in time and finite elements in space scheme using Gaussian quadrature integration. The solution to the iterative model is then found by Newton-Raphson iterations. This paper aims to verify and compare the new *Flow* model to existing models for vadose zone flow situations. Verification against an analytical solution was performed by comparisons of the maximum soil depth that could be maintained by a specified evaporative surface flux. Thereafter, the spatial and numerical scheme was compared to another numerical model (*Hydrus-1D*) implementing a backward Euler in time and finite difference scheme in space, solved by modified Picard iterations. Finally, the time stepping scheme and absolute run times were compared with the numerical model. The analysis showed that the results of *Flow* did not significantly differ from the solutions found by the analytical model. The spatial and temporal scheme comparisons did not result in systematic differences but sensitivities were found for both spatial discretization and time step size variations. Generally, the *Flow* model was observed to converge one order faster than the numerical model it was compared with, except where oscillatory *Flow* solutions were found. The time stepping schemes between the numerical models were observed to be similar while the absolute duration of simulations was found to be a factor 100 larger for the *Flow* model. It is concluded that the new *Flow* model passes verification with respect to the analytical solution and opens possibilities for further comparisons and justifies utilization which will enhance further development of the model in the hydrological field of vadose zone flow simulations.

1 Introduction

As for most situations no analytical solution to Richards' equation exists [Miller et al., 1998], numerical models will remain in use for solving unsaturated flow problems. Although many codes have been written to find numerical solutions to Richards' equation, no universal solution scheme performing equally accurate over the wide range of soils, boundary and initial conditions has been found [Farthing and Ogden, 2017], [Zha et al., 2019]. The unsaturated hydraulic conductivity function and soil water retention relation may parameterize the soil such that infinitely small and large values are applied simultaneously which unveils the degenerate property of Richards' equation [List and Radu, 2016]. This degeneracy induced by extremely nonlinear soil moisture relations has been the main issue for analysis and design decisions for numerical schemes for the Richards equation [Miller et al., 2013] where spatial and temporal discretization in combination with linear and nonlinear solvers have been investigated intensively in order to improve robustness, accuracy and speed of solutions [Farthing and Ogden, 2017], [Zha et al., 2019]. For advances in solving the Richards equation with respect to these points of interest, several directions of improvement should be investigated further. Some potential directions suggested in literature are eliminating the degenerate property of the Richards equation by reformulation of the equation, e.g. the Soil Moisture Velocity Equation (SMVE) [Ogden et al., 2017]. On the other hand further development with the focus on adaptive, in contrast to static spatial and temporal solution schemes, is suspected to be a potential direction of new advancements [Mostaghimi et al., 2015], [Baron et al., 2017]. Regardless of the theoretical approach for improvements, a large practical hurdle in advances of developments are the legacy languages (Fortran [Backus et al., 1957], C++ [ISO, 1995]) in which most models are written [Zha et al., 2019]. Python [Van Rossum and Drake, 2009] is an increasingly popular language with a relatively flat learning curve [Srinath, 2017] where memory management is implicitly performed by the language's bytecode itself. This would make use of the model more accessible to a larger group of researchers and might enhance the development of new approaches for solving unsaturated flow problems. In addition, the community is lacking a central database where consistent benchmarks and structured scheme comparison approaches have been standardized for the hydrological community to verify their models against [Farthing and Ogden, 2017]. The observation and prediction of the approximate doubling of transistors on a dense integrated circuit every two years, also know as Moore's Law [Brock and Moore, 2006], indicates that computational power becomes a less restricting factor which allows developers to choose from a wider range of programming languages where performance would have previously been a main obstacle.

The different and individually developed simulation software that exists nowadays is not limited to multidimensional and interdisciplinary models which simulate saturated and vadose zone flow problems; MODFLOW [McDonald and Harbaugh, 2003] is an example of such a model. However, for this study the interest is focussed on the one-dimensional models describing vadose or unsaturated zone flow. Although multidimensional models exist for vadose flow simulations, it is still considered relevant to develop one-dimensional models for these purposes due to the predominantly vertical direction of movement of flow in unsaturated soils. Additionally, extending such one-dimensional models is less complex and furthermore serves educational purposes.

Such a model, *Flow*, was developed for this study. This model, as opposed to existing models, is written in the Python programming language and will be compared with an existing model that has proven its performance and reliability. In table 1 a collection of popular and comparable models is presented. From table 1, *HYDRUS-1D* or *Hydrus* is selected and will be used for comparison to the new *Flow* model in this study. *SWAP* [Kroes et al., 2009] was developed at the University of Wageningen and *Hydrus* [Šimůnek et al., 2008] was developed at the University

Table 1: One-dimensional vadose zone models. All models are open source and the source code is written in Fortran except for *Flow*.

Model	Latest release	Scheme	Iteration procedure	Reference
Flow	2020	FE	Newton-Raphson	-
HYDRUS-1D	2014	FE	Picard	[Šimůnek et al., 2008]
MACRO	2010	FD	[Celia et al., 1990]	[Larsbo and Jarvis, 2003]
SWAP	2008	FD	Newton-Raphson	[Kroes et al., 2009]
WAVE	1996	FD	Newton-Rapshon	[Vanclooster et al., 1996]

of California Riverside. The latter model has non-open source two- and three-dimensional equivalents. *MACRO* [Larsbo and Jarvis, 2003] and *WAVE* [Vanclooster et al., 1996] were developed at the University of Agricultural Sciences in Sweden and at the University of Leuven in Belgium, respectively. The selection of any of these models is not preferred based on performance of water flow simulations as they compare with relatively small differences, although an error in the water balance of the *WAVE* model was found for some experiments [Vanderborght et al., 2005]. As no major differences between the models exist, *Hydrus* was chosen because it specifically focuses on the hydrological aspects of a simulation, in contrast to *SWAP* which also focuses on the vegetation and atmospherical interactions. Additionally, Hydrus allows for the implementation of different soil hydraulic functions (i.e. retention and conductivity), such as the model described by [R. H. Brooks & A. T. Corey, 1964]. Moreover, *Hydrus* has a graphical user interface (GUI) making it easier to use.

1.1 Research questions

The main purpose of this study is to examine how *Hydrus* and the new *Flow* model compare, both quantitatively and qualitatively for simulations of one-dimensional hydrological systems that describe flow in variably saturated soils. This main question is answered on the basis of three separate cases. The first case encapsulates the verification of the *Flow* model by comparison with an analytical solution for several different soil types. This is considered to be a critical step as large differences between model outcome and known analytical solutions would provide a weak basis for further analysis. In the second case the numerical solution schemes will be compared. This will provide insight in the model performances based on numerical scheme, drying and wetting soil simulations and the effect of change in nodal discretizations. Finally, the time complexity of *Hydrus* and *Flow* is compared, i.e the amount of time that is needed for both models to solve for a specific situation.

1.2 Theory of *Flow*

Mathematical Background The *Flow* model allows for saturated and unsaturated groundwater calculations. The governing flow equation passed to the model determines which situation will be simulated. For example, the flow equation of Darcy [Darcy, 1856] or Richards [Richards, 1931], the latter is sometimes referred to as Darcy-Richards equation, can be implemented for the simulation of saturated and unsaturated flow respectively. In this study the model simulations are focussed on unsaturated flow simulations only. The basic flow equation of Darcy [Darcy, 1856] in combination with the continuity equation defines the mathematical basis of the *Flow* model.

Assume the continuity equation that conserves the mass of a fluid that flows through a porous media:

$$\frac{\partial(\rho q_x)}{\partial x} + \frac{\partial(\rho q_y)}{\partial y} + \frac{\partial(\rho q_z)}{\partial z} + F = \frac{\partial(\rho q)}{\partial t} \tag{1}$$

The first three terms on the left hand side describe the flow in the x, y and z direction respectively. F represents additional sinks and sources. The right hand size of the equation defines the storage change in the system which defines its transient nature. The *Flow* model assumes isothermal, incompressible flow in a soil layer that is not elastic and simulates one-dimensional flow only. Assuming that the liquid density ρ is constant and omitting the y and z directions, eq. (1) can be written as follows:

$$\frac{\partial q_x}{\partial x} + F = \frac{\partial q}{\partial t} \tag{2}$$

Depending on the type of flow that will be simulated, q_x in eq. (2) is replaced by an equation that defines this type of flow. See eq. (3) for the implemented saturated flow situation

$$\frac{\partial}{\partial x}\left(k(x,H)\frac{\partial H}{\partial x}\right) + F = \frac{\partial H}{\partial t} \tag{3}$$

where H is the hydraulic head and $k(x,H)$ a hydraulic conductivity function. To transform eq. (3) to a situation that describes vertical flow in an unsaturated system, the orientation of the model is rotated counter clockwise for 90 degrees, see fig. 2. Additionally, the hydraulic head H is represented as the sum of the pressure head h and a specific reference level z,

$$\frac{\partial}{\partial z}\left(k(z,h+z)\frac{\partial(h+z)}{\partial z}\right) + F = \frac{\partial \theta}{\partial t} \tag{4}$$

which can be written as:

$$\frac{\partial}{\partial z}\left(k(z,h+z)\left(\frac{\partial h}{\partial z} + \frac{\partial z}{\partial z}\right)\right) + F = \frac{\partial \theta}{\partial t} \tag{5}$$

Assuming z to be zero at the groundwater level results in the following unsaturated flow equation as first described by [Richards, 1931] which is also known, ignoring the sources and sinks term F, as the mixed form of the Richards equation.

$$\frac{\partial}{\partial z}\left(k(z,h)\left(\frac{\partial h}{\partial z} + 1\right)\right) + F = \frac{\partial \theta}{\partial t} \tag{6}$$

Note that the storage change of the saturated and unsaturated flow equations are defined in terms of hydraulic head H and moisture content θ respectively. The governing flow equation, the part that was substituted for q_x, will be indicated with Q in the remaining part of this study.

Numerical Scheme A finite elements scheme is used to formulate a system of linear equations based on the spatial components of eq. (6) that is solved using the Newton-Raphson method [Ypma, 1995]. This is an iterative solving method that needs initial state values of the system as a first estimate of the current state of the system. A Newton-Raphson iteration is defined as follows:

$$A_{\tau-1} \times y_\tau + F_{\tau-1} = 0 \tag{7}$$

where A is the Jacobian matrix of size $[N \times N]$ and F a vector of size $[1 \times N]$ that contains the sum of all forcing fluxes on the system including the storage change flux. N holds the total number of nodes in the system for which the model is solved. The matrix equation is solved for

y using Gaussian elimination [Atkinson, 2008] and the result is added to the initial states, s_0, or the states of the previous iteration.

$$s_\tau = s_{\tau-1} + y_\tau \tag{8}$$

The above procedure described in eq. (7) and eq. (8) is repeated until the system has been converged; subscript τ indicates the iteration number. Convergence is achieved when the maximum change between any of the nodes no longer exceeds a set threshold. The exact definition is:

$$max(abs(s_{\tau-1} - s_\tau)) < max(abs(\epsilon * s_\tau)) \tag{9}$$

where ϵ represents the maximum change as fraction of the value allowed at any of the nodes.

Integration of functions, e.g. the governing flow equation Q or spatial forcing functions, over the nodal segments is performed using the Gaussian quadrature method [Abramowitz and Stegun, 1948]. This is needed to calculate the contribution of a specific forcing towards the neighboring nodes and to calculate the components of the Jacobian matrix that is core to the numerical Newton-Raphson solve procedure. The Jacobian matrix A contains all derivatives of the flux functions in the system that depend on the states. This matrix can be defined as the sum of its individual components.

$$A = A_{sys} + A_{spat} + A_{point} \tag{10}$$

where A_{sys} contains the derivatives of the governing flow equation Q. A_{spat} and A_{point} contain the derivatives of the state dependent spatial and point forcing flux functions respectively. The components of the governing flow equation Q that result in A_{sys} are calculated as follows, assuming that Q takes position χ, state s and gradient of state ψ as its three arguments.

$$\frac{\partial Q}{\partial s}_i = \sum_{\lambda=1}^{\Lambda} \frac{Q(\chi_{i,\lambda}, s_{i,\lambda} + \partial s, \psi(L_i, s + \partial s)) - Q(\chi_{i,\lambda}, s_{i,\lambda}, \psi(L_i, s))}{\partial s} * w_\lambda \tag{11}$$

where the state value s at position χ is calculated by linear interpolation between the nearest known states

$$s_{i,\lambda} = s_i * (1 - p_\lambda) + s_{i+1} * p_\lambda \tag{12}$$

and the gradient ψ is the change of state over the length of nodal segment

$$\psi(L_i, s) = \frac{s_{i+1} - s_i}{L_i} \tag{13}$$

where

$$L_i = x_{i+1} - x_i \text{ for } i = 1, 2, \ldots, N-1 \tag{14}$$

i and λ are integer indices starting at one indicating the selected node and Gaussian quadrature degree respectively. χ and p both represent the position of the Gaussian quadrature point with the only difference being that χ holds the absolute position with respect to the complete flow domain x, and p holds the relative position with respect to the segment between the adjacent nodes x_i and x_{i+1}. The corresponding Gaussian quadrature weights are presented by ω, see fig. 1 for a visualization of the flow domain which includes a schematic of Gaussian integration for degree $\Lambda = 3$. The resulting derivatives as calculated by eq. (11) are substituted in the

Jacobian matrix which results in the following sparse matrix for the governing flow equation Q:

$$A_{sys} = \begin{bmatrix} -\frac{\partial Q}{\partial s}_i & -\frac{\partial Q}{\partial s}_i & & & & \\ \frac{\partial Q}{\partial s}_i & \frac{\partial Q}{\partial s}_i - \frac{\partial Q}{\partial s}_{i+1} & -\frac{\partial Q}{\partial s}_{i+1} & & & \\ & \frac{\partial Q}{\partial s}_{i+1} & \frac{\partial Q}{\partial s}_{i+1} - \frac{\partial Q}{\partial s}_{i+2} & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \frac{\partial Q}{\partial s}_{N-1} - \frac{\partial Q}{\partial s}_N & -\frac{\partial Q}{\partial s}_N \\ & & & & \frac{\partial Q}{\partial s}_N & \frac{\partial Q}{\partial s}_N \end{bmatrix} \tag{15}$$

As mentioned above, the *Flow* model also allows for miscellaneous spatial forcing functions that are dependent on the state of the system itself. These forcing functions are indicated with S_j where the index j identifies the selected spatial forcing function since multiple forcing functions can be implemented. Functional description of root water extraction by plants or the storage change function itself are examples of such state dependent functions that can be implemented in the system. S takes position χ and state s as its consecutive arguments. The calculation of the components that contribute to the Jacobian matrix A_{spat} is defined as follows:

$$\sum \frac{\partial S_l}{\partial s}_i = \sum_{\lambda=1}^{\Lambda} \sum_{j=1}^{n} \frac{S(\chi_{i,\lambda}, s_{i,\lambda} + \partial s)_j - S(\chi_{i,\lambda}, s_{i,\lambda})_j}{\partial s} * L_i * w_\lambda * (1 - p_\lambda) \tag{16}$$

and

$$\sum \frac{\partial S_r}{\partial s}_i = \sum_{\lambda=1}^{\Lambda} \sum_{j=1}^{n} \frac{S(\chi_{i,\lambda}, s_{i,\lambda} + \partial s)_j - S(\chi_{i,\lambda}, s_{i,\lambda})_j}{\partial s} * L_i * w_\lambda * p_\lambda \tag{17}$$

The approach used to calculate the components can be thought of as a loop in a loop. The outer loop holds the first step of the Gaussian quadrature method. The inner loop accumulates all the spatial forcing functions n over the currently selected integration step of the outer loop. This procedure continues until the outer loop terminates in Λ steps, see fig. 1 for a visualization of the Gaussian quadrature method. The calculation of the state $s_{i,\lambda}$ is defined in eq. (12) above. The subscript l and r in eq. (16) and eq. (17) indicate the distribution of the sum of all components calculated from the spatial state dependent forcing functions to the left and right node of the current segment respectively. These calculated components are substituted in the

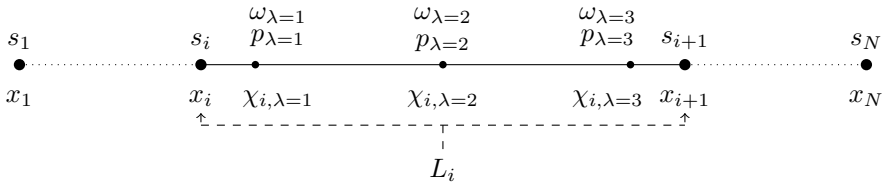


Figure 1: Schematic of a segment between two neighboring nodes at which contributions (fluxes or derivatives for the Jacobian) to the states s at nodes x are calculated using the Gaussian quadrature method [Abramowitz and Stegun, 1948] of degree $\Lambda = 3$. χ, p and ω represent the absolute and relative position and corresponding weights respectively. The segment length is indicated with L_i and N denotes the total amount of nodes.

Jacobian matrix A_{spat} which results in the following sparse matrix:

$$A_{spat} = \begin{bmatrix} \Sigma\frac{\partial S_l}{\partial s}_i & \Sigma\frac{\partial S_r}{\partial s}_i & & & & \\ \Sigma\frac{\partial S_l}{\partial s}_i & \Sigma\frac{\partial S_r}{\partial s}_i + \Sigma\frac{\partial S_l}{\partial s}_{i+1} & \Sigma\frac{\partial S_r}{\partial s}_{i+1} & & & \\ & \Sigma\frac{\partial S_l}{\partial s}_{i+1} & \Sigma\frac{\partial S_r}{\partial s}_{i+1} + \Sigma\frac{\partial S_l}{\partial s}_{i+2} & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \Sigma\frac{\partial S_r}{\partial s}_{N-1} + \Sigma\frac{\partial S_l}{\partial s}_N & \Sigma\frac{\partial S_r}{\partial s}_N \\ & & & & \Sigma\frac{\partial S_l}{\partial s}_N & \Sigma\frac{\partial S_r}{\partial s}_N \end{bmatrix} \tag{18}$$

The last function class that contributes to the Jacobian matrix A are the point flux functions that depend on the state of the system. These forcing functions are denoted with P_k where k indicates a selected function out of m forcing functions that the model incorporates. Examples of such functions are extraction of a well in saturated conditions or any artificial injection of water in an unsaturated soil at a specific point. P takes state s as its only explanatory argument as the point flux at a different position is just another point flux without spatial dependence. This position is contained in the index parameter k. See eq. (19) below for the definition of the calculation that defines the individual components that contribute to the Jacobian matrix A_{point}:

$$\sum \frac{\partial P}{\partial s}_i = \sum_{k=1}^{m} \frac{P(s_{i,k} + \partial s)_k + P(s_{i,k})_k}{\partial s} \tag{19}$$

where all symbols have been defined above. In the situation of the spatial dependent point flux function there is no need for Gaussian quadrature integration because of the one-dimensional nature of the state dependent point flux function. The calculation of state $s_{i,k}$ at the position of the state dependent point flux is calculated as described by eq. (20):

$$s_{i,k} = s_i + rfac_k * (s_{i+1} - s_i) \tag{20}$$

In eq. (21) the distribution of the derivatives calculated by eq. (19) towards the left and right node of the current segment is defined. A segment lays between two neighboring nodes or points at which the system will be solved.

$$\begin{aligned} \sum \frac{\partial P_l}{\partial s}_i &= \sum \frac{\partial P}{\partial s}_i * lfac_k \\ \sum \frac{\partial P_r}{\partial s}_i &= \sum \frac{\partial P}{\partial s}_i * rfac_k \end{aligned} \tag{21}$$

Distribution of fluxes towards the nearest two nodes is determined by the fractions $lfac_k$ and $rfac_k$ for the left and right node respectively. The fractions sum to one and describe the relative position of the point flux function on the nodal segment. The individual components that result from eq. (21) are substituted in A_{point} which results in the diagonal matrix below.

$$A_{point} = \begin{bmatrix} \Sigma\frac{\partial P_l}{\partial s}_i & & & & \\ & \Sigma\frac{\partial P_r}{\partial s}_i + \Sigma\frac{\partial P_l}{\partial s}_{i+1} & & & \\ & & \Sigma\frac{\partial P_r}{\partial s}_{i+1} + \Sigma\frac{\partial P_l}{\partial s}_{i+2} & & \\ & & & \ddots & \\ & & & \Sigma\frac{\partial P_r}{\partial s}_{N-1} + \Sigma\frac{\partial P_l}{\partial s}_N & \\ & & & & \Sigma\frac{\partial P_r}{\partial s}_N \end{bmatrix} \tag{22}$$

The three separate Jacobian matrices as defined above result in the complete Jacobian matrix A by summing of the elements of the separate parts, see eq. (10). The other component that is needed for the Newton-Raphson iteration method, eq. (7), are the actual flux values itself, collected in forcing vector F. This forcing vector can be separated as follows:

$$F = F_{internal} + F_{external} \tag{23}$$

where $F_{internal}$ holds the internal fluxes which are inherent to the system's governing flow equation Q and internal gradients in the system itself. Calculation of the individual flux components at the nodes in the flow domain is calculated as described by eq. (24):

$$F_i = \sum_{\lambda=1}^{\Lambda} Q(\chi_{i,\lambda}, s_{i,\lambda}, \psi_{L_i,s}) * \omega_\lambda \tag{24}$$

where the symbols are as used above. The components defined by eq. (24) are distributed in $F_{internal}$ according to the following format:

$$F_{internal} = \begin{bmatrix} -F_i \\ F_i - F_{i+1} \\ F_{i+1} - F_{i+2} \\ \vdots \\ F_{N-1} - F_N \\ F_N \end{bmatrix} \tag{25}$$

The calculation of the internal fluxes is essential for the *Flow* model and is therefore automatically included and cannot be removed. In contrast to the internal fluxes, all other forcing fluxes applied to the system are optional and are collected in $F_{external}$. The optional forcing or external forcing can be separated into two groups: spatial fluxes and point fluxes. Spatial forcing is introduced using spatial forcing functions where the calling signature of the function defines its dependency on position, state or both. Besides, spatial fluxes can also be added to the system in array or scalar form when independent from state s. To calculate the contribution of the spatial forcing functions to the system's forcing vector $F_{external}$, the individual components are calculated as shown in eq. (26) where subscript l and r indicate the nearest left and right node respectively:

$$\begin{aligned} F_i &= \sum_{\lambda=1}^{\Lambda} S_j * \omega_\lambda * L_i \\ F_{l_i} &= F_i * (1 - p_\lambda) \\ F_{r_i} &= F_i * p_\lambda \end{aligned} \tag{26}$$

S_j is any of the spatial forcing functions in $\{S(\chi_{i,\lambda}, s_{i,\lambda}), S(s_{i,\lambda})\}$, see above. Arrays or scalar values that contain spatial forcing data are added to $F_{external}$ by multiplication of the value(s) with an adapted form of the length vector L. The adapted length vector is calculated as follows:

$$l_i = \begin{cases} \frac{x_i + x_{i+1}}{2} - x_i & \text{for } i = 1 \\ \frac{x_{i+1} - x_{i-1}}{2} & \text{for } i = 2, 3, \ldots, N-1 \\ x_i - \frac{x_{i-1} + x_i}{2} & \text{for } i = N \end{cases} \tag{27}$$

where the number of lengths is not equal to the number of segments as in eq. (14), but is equal to the number of nodes in the system which allows for direct accumulation to the $F_{external}$ forcing vector after multiplication with the array or scalar spatial forcing value(s).

The other group of forcing fluxes, the point fluxes, can be implemented in the model in the form of a state dependent forcing function or as a scalar value. The calculation of the individual components that contribute to the external forcing vector $F_{external}$ are both calculated by the components described by eq. (28).

$$\begin{aligned} F_{l_i} &= P_k * lfac \\ F_{r_i} &= P_k * rfac \end{aligned} \tag{28}$$

where P_k is substituted with a point forcing function or scalar value, $\{P(s_{i,k}), P\}$. The components that are calculated from the spatial forcing S_j and point forcing P_k, eq. (26) and eq. (28) respectively, are accumulated to the external forcing vector $F_{external}$ according to the following format:

$$F_{external} = \begin{bmatrix} F_{l_i} \\ F_{r_i} + F_{l_{i+1}} \\ F_{r_{i+1}} + F_{l_{i+2}} \\ \vdots \\ F_{r_{N-1}} + F_{l_N} \\ F_{r_N} \end{bmatrix} \tag{29}$$

The added internal and external flux, eq. (23), represent the total forcing on the system. This concludes the description of the Newton-Raphson method and the definition of its components. Note the similarities between the calculation of the components for the Jacobian matrix A and the calculation of specific forcing fluxes for the forcing vector F. Not all spatial forcing calculations require the use of the Gaussian quadrature method for integration and the forcing flux calculations are performed individually for each forcing flux implemented. This in contrast to the calculation of the Jacobian matrix components for the state dependent spatial forcing functions which are calculated in one go by accumulation of the forcing fluxes before integration over the (partial) segment, see eq. (16) and eq. (17).

Boundary Conditions Two different types of boundary conditions can be implemented in the system. A boundary condition of type Dirichlet, BC_D, which sets a fixed state, or a boundary condition of type Neumann, BC_N, which sets a fixed flux. Both edges of the flow domain default to BC_D; however any combination of boundary conditions is allowed except for two BC_N boundary conditions because that would lead to a trivial system with infinite solutions that only holds relative relations as the system is not fixed to an absolute point of reference.

The boundaries of *Flow* have several different names depending on the type of flow that is modelled. See fig. 2 that defines the names of the specific orientations. Unsaturated flow uses the names bottom and top which map to the indices 1 and N in the Newton-Raphson equation and eq. (7).

The implementation of the Dirichlet or fixed state boundary condition is straightforward. Assume the implementation of such a boundary condition at the bottom of the flow domain. Row 1 is excluded from the matrix equation described in the Newton-Raphson method and by eq. (7) and the value of BC_D is directly set to the first index of state vector at s_τ, see eq. (8). No solution is calculated for y_1 due to the exclusion of row 1 from the matrix equation.

Assume the implementation of a Neumann or fixed flux boundary condition at the top of the domain. In this case, row N is not excluded from the matrix equation described in the

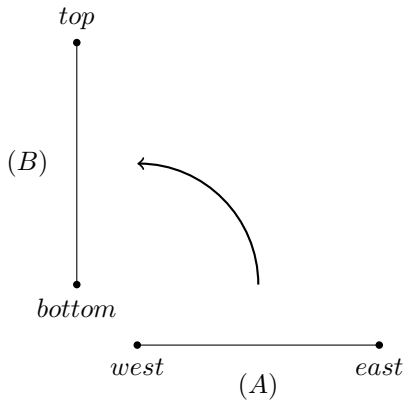


Figure 2: Orientation of the model domain for saturated (A) and unsaturated (B) flow simulations and the corresponding names that are assigned to the boundaries.

Newton-Raphson method and by eq. (7) but the boundary condition flux value BC_N is added to the forcing vector element F_N on each iteration. See eq. (30) that shows how the matrix equation of a Newton-Raphson iteration is defined for a bottom and top boundary condition of type Dirichlet and Neumann respectively.

$$\begin{bmatrix} A_{2,1} & \cdots & A_{2,N} \\ \vdots & \ddots & \vdots \\ A_{N,1} & \cdots & A_{N,N} \end{bmatrix} \times \begin{bmatrix} BC_D \\ y_2 \\ \vdots \\ y_N \end{bmatrix} + \begin{bmatrix} F_2 \\ \vdots \\ F_N + BC_N \end{bmatrix} = 0 \tag{30}$$

Note that the first row of the coefficient matrix A and forcing vector F is excluded from the matrix multiplication and addition respectively for the implementation of the Dirichlet boundary condition BC_D. The implementation of the Neumann boundary condition BC_N only adds an additional term to the forcing vector F.

Transience of the System As already mentioned, the transience of the system is implemented in the *Flow* model by inclusion of the transient term $\frac{\partial q}{\partial t}$ from eq. (2) in the forcing vector F. The storage change is integrated into the system as a spatial state dependent forcing function and thus also contributes to the components of the Jacobian matrix A. The definition of the storage change function for saturated flow from eq. (3) is presented below:

$$\frac{\partial H}{\partial t} = \frac{\partial s}{\partial t} = -f_{st} * \frac{s_{i,t} - s_{i,t-1}}{\Delta t} \tag{31}$$

where is hydraulic head H is assumed to be equal to the pressure head or state as denoted by s. f_{st} is the storativity fraction and the time step size is indicated with Δt. The unsaturated equivalent of the storage change function, see eq. (6), is defined in a similar way but note the essential differences in eq. (32).

$$\frac{\partial \theta}{\partial t} = \frac{\theta_t - \theta_{t-1}}{\Delta t} = -\frac{fun(s_{i,t}) - fun(s_{i,t-1})}{\Delta t} \tag{32}$$

The difference in state values is calculated as difference in volumetric water content by fun which is a function that describes the pF-curve of the current soil, i.e. soil water retention curve.

Implementation of the storage change function, eq. (31) or eq. (32), is optional and when omitted the *Flow* model will describe a stationary system. This stationary system is solved using the Newton-Raphson iteration procedure as described by eq. (7) and eq. (8), until the conversion threshold is met, see eq. (9). The transient system, when a storage change function is implemented, is solved by concatenation of solutions from the individual transient solve procedures, see fig. 3. The Newton-Raphson iteration can be seen as an inner loop that is executed over a time step until the complete simulation time $tMax$ is reached. Repetition over the time steps is characterized as the outer loop.

The size of the time step over which the inner loop is performed is not necessarily constant and the *Flow* model can be solved by less outer loop iterations if the time step size is increased. See fig. 4 that shows a more detailed representation of fig. 3 which also includes an algorithm that decides on the time step size of the outer loop. This is the essential transient solve procedure and is referred to as the 'variable time block'.

To demonstrate the process executed for the transient solve procedure, a complete cycle through the process is described. The starting point is at the "IN" node in fig. 4. A formulated *Flow* model should at least contain a governing flow equation Q and the model domain x should be discretized in which boundary conditions are set. The system is transient in this example, so a storage change function should be implemented too. Initial states are automatically created at initialization of the model but may be modified. After sufficiently defining the initial model the first green decision block is entered. Assume that the decision evaluates to *true* so the green arrow is followed and the *dt_solve* block is entered, see fig. 5a. Input to this procedure is the previous or initial time step dt. This procedure returns the number of iterations as a result of convergence or due to exceeding the threshold value $MaxIt$. Progressing from the purple block in fig. 4 to the next green decision block and assuming that the *dt_solve* method returned due to convergence, the dashed red arrow is followed to the next purple node *dt_block*, see fig. 5b. This block takes current time t, the current time step dt and the number of iterations IT that was returned by the previous *dt_solve* block and calculates on the basis of these values whether the time step dt should increase, decrease or remain the same for the next cycle through the 'variable time block', fig. 4. Increase or decrease of the time step is calculated by multiplication with the constants $DMul$ or $DMul2$ respectively. Decision of the time step adaptation is chosen based on limits set on the number of iterations IT from the previous *dt_block* block ($ItMin$ & $ItMax$) or due to limitations set upon the time step dt itself ($dtMin$ & $dtMax$). This *dt_block* block updates the current temporal position of the model with the old time step and returns the new time t as well as the new time step dt. On return of the second purple block *dt_block* a full cycle through the 'variable time block', fig. 4, is completed and can be repeated as many times as needed. For the context of the 'variable time block' see fig. 17 where the complete schematic of the solve procedure is presented.

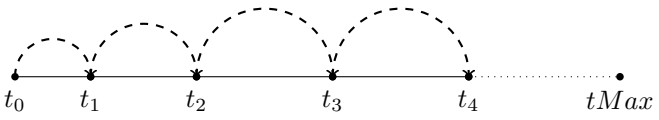


Figure 3: Schematic of transient solve procedure loops. The arrows represent a Newton-Raphson iteration procedure over a time step (inner loop). This procedure is repeated over the time steps until $tMax$ is reached (outer loop). Note the variable time step over which the inner loop is performed.

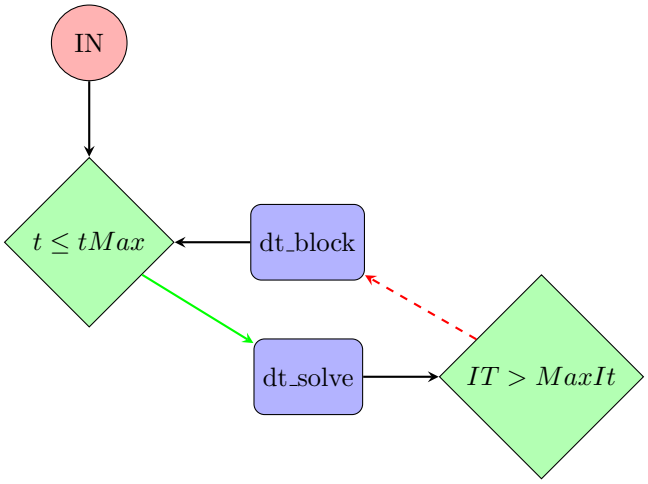
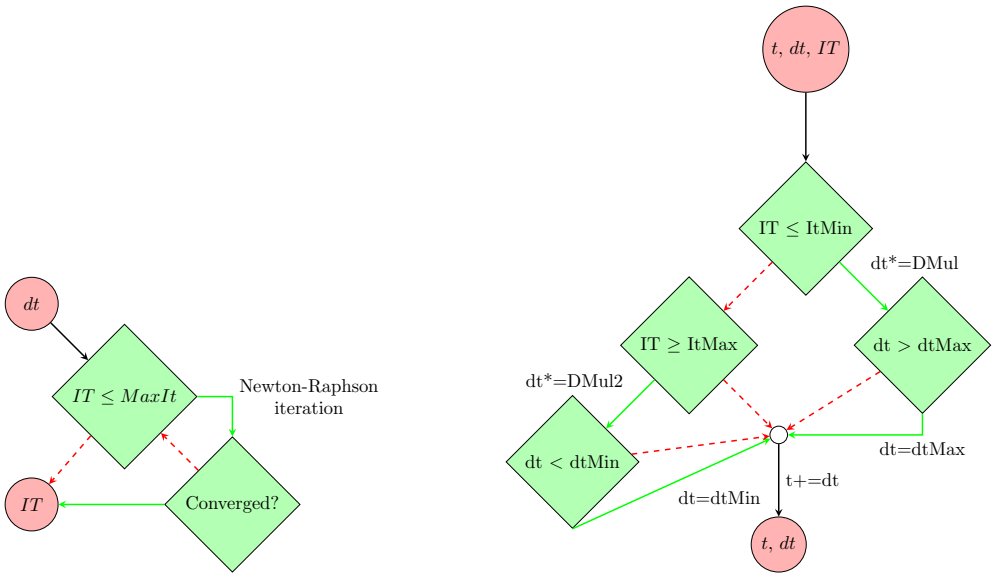


Figure 4: The variable time block consists of two parts: *dt_solve* and *dt_block*. In the first part the Newton-Raphson iteration or inner loop is executed. The last part decides on the time step size *dt* for the next cycle in the variable time block procedure. One complete cycle in the variable time block is the equivalent of one outer loop iteration. Arrows indicate the direction of flow in the diagram when conditions evaluate positive (green) or negative (red).



(a) dt_solve - Calculates the next model solution over the current time step *dt* using the Newton-Raphson procedure [Ypma, 1995].

(b) dt_block - Procedure that calculates the time step size *dt* as a function of time step size itself and the number of iterations IT.

Figure 5: dt_solve and dt_time blocks from the 'variable time block' in fig. 4. Arrows indicate the direction of flow in the diagram when conditions evaluate positive (green) or negative (red).

Water Balance The water balance over a specific depth is an important output and is included in the *Flow* model to check on numerical deficiencies and to create a record of the different fluxes distributed, which allows for a close inspection on the model. The water balance is defined as follows:

$$F_{net} = F_{internal} + F_{external} = F_{internal} + \sum_{j=1}^{n} S_j + \sum_{k=1}^{m} *P_k \tag{33}$$

F_{net} is the net flow in the system and should be within tolerance, it describes the numerical mismatch after each Newton-Raphson iteration that has been terminated on its convergence threshold. All other symbols in eq. (33) are as described above.

See the *Glossary* in the appendix for a list of all parameters, their meaning and their definitions in the *Flow* model.

2 Materials & Methods

In this study a structured analysis of the *Flow* model will be conducted. This analysis is structured using three separate cases which contain the verification and comparison of the *Flow* model using two different other models. The cases are structured such that they derive from a base case. Parametrization for each case, including the base case, is presented in tabulated form. For this analysis an analytical [Malik et al., 1989] and numerical model [Šimůnek et al., 2008] will be used. A short description of the latter numerical model is included in this section of the text. In addition, practical implementation and comparison methods are discussed to provide insight in the procedures used during the execution of this study.

Hydrus *Hydrus* is a software package that is used for the simulation of one-dimensional flow of water, heat and multiple solutes in soils ranging from unsaturated to completely saturated soils including horizontal, vertical and inclined flow directions, see [Šimůnek et al., 2013] for the complete description of the model. In this study *HYDRUS-1D* version 4.17 is used, see [Šimůnek et al., 2008] for the historical development of this software package and its complete feature set. Only complementary components of the *Hydrus* model that are also implemented in the *Flow* model are used in this comparison study. This includes the isothermal vertical flow of water in homogeneous soils of variably saturated water contents but excludes saturated flow. The governing flow equation used for the simulation of the unsaturated flow is Richards' equation [Richards, 1931] and the hydraulic model, soil water retention curve $\theta(h)$ and unsaturated hydraulic conductivity function $k(h)$, as described by [van Genuchten, 1980] is used. *Hydrus* also allows for other hydraulic models such as the model described by [R. H. Brooks & A. T. Corey, 1964] or even dual porosity or dual permeability models. Below a short technical description of the relevant technical implementation of the *Hydrus* model is presented.

Mathematical Background *Hydrus* [Šimůnek et al., 2013] implements the mixed form of Richards' equation because of performance reasons [Celia et al., 1990], see eq. (34).

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial x}\left[K(h,x)\left(\frac{\partial h}{\partial x} + cos(\alpha)\right)\right] - S \tag{34}$$

The direction of unsaturated flow in this study is strictly vertical which reduces the above equation to eq. (35):

$$\frac{\partial \theta}{\partial t} = \frac{\partial}{\partial x}\left[K(h,x)\left(\frac{\partial h}{\partial x} + 1\right)\right] - S \tag{35}$$

This equation is identical to eq. (6) despite the use of slightly different symbology.

Numerical Scheme The *Hydrus* model implements a mass-lumped linear finite elements scheme to the mixed form of the Richards equation and solves the system of linear equations using the Picard iteration scheme [Celia et al., 1990]. Because the manual of *Hydrus* [Šimůnek et al., 2013] lacks a detailed description of the derivation of its numerical scheme used to solve Richards' equation, a more detailed description of the procedure will be discussed in this paragraph. Note that the external forcing term S from eq. (35) is omitted in the below derivation.

The mass-lumped form of eq. (35) reads as follows and is essentially equal to the implicit Euler discretization of the time derivative, [Celia et al., 1990]:

$$\frac{\theta^{n+1} - \theta^n}{\Delta t} - \nabla \cdot K^{n+1} \nabla H^{n+1} - \frac{\partial K^{n+1}}{\partial z} = 0 \tag{36}$$

where the superscript n defines the time level. Now Richards' equation is written in the implicit form, the picard iteration scheme can be implemented. The implemented scheme looks as follows:

$$\frac{\theta^{n+1,m+1} - \theta^n}{\Delta t} - \nabla \cdot K^{n+1,m} \nabla H^{n+1,m+1} - \frac{\partial K^{n+1,m}}{\partial z} = 0 \tag{37}$$

where the level of iteration is denoted by m. As mentioned in [Šimůnek et al., 2013], this form of the implemented picard iteration scheme in the Euler implicit or mass-lumped form of Richards' equation contains two different variables that need to be known at the next level of iteration m, namely θ and H. To overcome this problem the term $\theta^{n+1,m+1}$ is estimated by a Taylor series expansion with respect to H. See the equation below:

$$\theta^{n+1,m+1} = \theta^{n+1,m} + C^{n+1,m}(H^{n+1,m+1} - H^{n+1,m}) \tag{38}$$

where $C = \frac{\partial \theta}{\partial h}$. This equation rewrites $\theta^{n+1,m+1}$ in terms of $H^{n+1,m+1}$ and is thereafter substituted into eq. (37) and rewritten in terms of increment in iteration level, to yield

$$\begin{gathered} -C^{n+1,m}\frac{H^{n+1,m} - H^n}{\Delta t} + \nabla \cdot K^{n+1,m} \nabla H^{n+1,m} + \frac{K^{n+1,m}}{\partial z} - \frac{\theta^{n+1,m} - \theta^n}{\Delta t} \\ = \\ R^{n+1,m}_{ModifiedPicard} \end{gathered} \tag{39}$$

This general mixed form is also called the *modified Picard approximation*, [Celia et al., 1990]. This scheme is general in the sense that it has no spatial discretization implemented yet. In *Hydrus* a finite elements scheme is implemented which is equivalent to the standard finite difference scheme [Vogel et al., 1996] where Gaussian quadrature integration [Abramowitz and Stegun, 1948] of degree one is applied, assuming $K_{1/2}$ being calculated as the arithmetic mean. The implementation of the finite elements scheme into eq. (39), after rewriting and shifting some terms reads as follows:

$$\begin{aligned} [H_{i-1}K_{i-1/2} + H_i(-K_{i-1/2} - K_{i+1/2} - C_i^{n+1,m}\frac{\Delta z^2}{\Delta t}) + H_{i+1}K_{i+1/2}]^{n+1,m} = \\ -\Delta z[K_{i+1/2} - K_{i-1/2}]^{n+1,m} + \frac{\Delta z^2}{\Delta t}(\theta_i^{n+1,m} - \theta_i^n - C_i^{n+1,m}[H_i]^n) \end{aligned} \tag{40}$$

The compact form of eq. (40) reads as:

$$A \cdot [H_i]^{n+1,m} = B \tag{41}$$

This latter equation finalizes the development of the mass-lumped linear finite elements scheme that *Hydrus* uses to solve Richards' equation for variably saturated media. See *Appendix - Hydrus schemes* for the complete derivation of equation eq. (41) where the equivalent derivations for the h-based Richards' equation are also presented to provide for easy comparison between the different schemes.

The scheme of equation eq. (41) is solved by picard iteration, [Coddington and Levinson, 1955], which evaluates the matrix equation by solving for the pressure head $H^{n+1,m}$ by Gaussian elimination, [Atkinson, 2008]. Each successive picard iteration uses the result of the previous iteration:

$$A^{n+1,m} \times H^{n+1,m} = B - S \tag{42}$$

the extra term S denotes the external forcing term which is calculated at the previous time level. This procedure needs initial conditions and is considered to be converged when the absolute change between successive iterations does not exceed a specific tolerance set.

Boundary Conditions *Hydrus* implements fixed state and flux boundary conditions into its model. In contrast with the *Flow* model, the fixed state or Dirichlet boundary condition is implemented in the existing solve scheme without cutting off the top or bottom row. The corresponding diagonal component of the coefficient matrix A, see eq. (41), is set to unity and the right hand side of the equation equals the value of the fixed boundary condition which leave the corresponding state, H_0 or H_N, to be equal to the value of the fixed boundary condition set on the right hand side of the equation after solving the matrix equation on that row.

Although *Hydrus* also implements several other types of boundary conditions such as the simulation of infiltration and surface ponding and free drainage at the bottom of the domain, here the focus is on state independent boundary conditions because only that type of boundary condition is implemented in the *Flow* model. *Hydrus* implements the fixed flux or Neumann boundary condition by discretization of the mass balance equation instead of Richards' equation because the latter can quickly lead to relatively unstable solutions [Šimůnek et al., 2013] when the boundary condition shows a strong variation in time. However, in this study only single temporal forcing periods are simulated during the comparison of the models. See the [Šimůnek et al., 2013] for the exact implementation and other details about the boundary conditions.

Transience of the System The size of the time step over which the model is solved is initially set to a small value. The size of the time step is then adjusted as a function of the progress the model makes when solving for the states. The exact procedure is developed by [Šimůnek et al., 1992] and [Mls, 1982] and is also explicitly described in [Šimůnek et al., 2013]. The numbers mentioned for the procedure in the manual are not fixed and can be altered to align with the implementation of the *Flow* model which has a comparable transient time step control system.

Water Balance While the *Flow* model calculates the water balance solely in terms of fluxes, the *Hydrus* model calculates the water balance in terms of volumes and fluxes. The total volume change over a certain time step should be equal to the flux over the boundaries of the system during that same time step. The external fluxes are included in the water balance in terms of volumes, see [Šimůnek et al., 2013] for the exact calculation and implementation of the water balance.

Comparison methods *Hydrus* is written in Fortran and the project is open source. *Hydrus* is designed to be used by its graphical user interface (GUI). It is also possible to run the model using the command line interface (CLI). In this study, input is passed to *Hydrus* using the command line interface. Making use of the GUI would be cumbersome and error-prone. Unfortunately, the CLI is not user friendly and it was needed to write a parser in order to submit the data to the *Hydrus* model. The *SELECTOR.IN* file is filled out with basic information, water flow information and time information. The discretization and initial states of the soil profile are passed to the model using a file called *PROFILE.DAT*. Thereafter the calculation of the model is executed using the *H1D_CALC.exe* binary for a direct solution. Those two input files do not include all the options that can be passed to *Hydrus* but it does include all input of which a complementary option is available in the *Flow* model and is therefore sufficient for this model comparison study. See [Šimůnek et al., 2013] for the complete description of all input files and executable binaries. The results of the *Hydrus* calculations are written to plain text files which are read and imported into Python in order to compare the results with the native Python *Flow* model.

Development of *Flow* The software package *Flow* is written in Python version 3.6, [Van Rossum and Drake, 2009]. Many additional modules are used in the implementation of the model, see table 15 in the appendix for the complete list of dependencies. The model has been tested on a Windows machine. No fundamental changes, if any, are expected to be needed in order to run the software on a Unix based system, e.g. Mac or Linux.

The directory structure of the software project presented in fig. 6 shows the structure of *Flow* as a Python Package. The main model is contained in `./flow/flow1d/flowFE1d.py`, the blueprint of the object oriented model is defined here. Many instances can be active in the same runtime environment. This is convenient in order to compare different model results at runtime or to perform batch calculations. Several external functions are included in `./flow/utility/*`. Functional relations that describe the soil hydraulic functions are contained in `conductivityfunctions.py`. Governing flow equations for saturated and unsaturated flow are stored in `fluxfunctions.py`, the storage change function that allows for transient flow calculations is also stored here. The file `helper.py` holds several miscellaneous functions, one important function defines the convergence criteria for the model. Data visualization functions are stored in `plotting.py`, and `spacing.py` contains functions that are able to calculate structured and unstructured nodal relations which are used for *Flow's* spatial discretization. Data needed for the parametrization of the model is stored in `./data/StaringReeks.txt`. The `__init__.py` files are required for the directory structure to be recognized by the system as a Python Package and may hold any arbitrary information about the model, e.g. the root location or efficient imports. The model is structured such that it is not limited to currently implemented features but allows for extensions. For example, different flow equations, conductivity and soil water retention or convergence functions can be used as long as the function signatures and argument types comply with the existing formats.

The *Flow* model can be installed using a Python package manager or from source. The source code and documentation [Berendsen, 2021] is hosted on GitHub where more information about both installation procedures is available. The project's documentation is generated using Sphinx

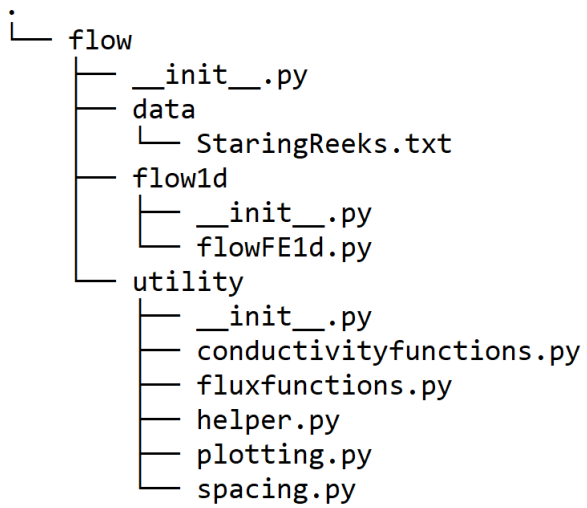


Figure 6: Directory structure of the *Flow* software model where the dot (.) indicates the root or current working directory.

[Brandl, 2010] and contains the application programming interface (api) documentation.

Implementation of the model has been conducted using an object oriented approach, as opposed to a functional approach. The main difference between these two approaches is that in an object oriented approach the state of the system is held within an object that is active at runtime. The functional programming style does not allow for holding the states of the system but passes arguments around in functions where there is generally an one-to-one relationship between input and output because functional procedures are not dependent on the internal state of the system. The properties of object oriented programming are used to store the state of the one-dimensional *Flow* model in objects at runtime which may also be written to disk for data persistence.

A test driven development (TDD) strategy [Beck, 2003] has been used for writing the source code of *Flow*. The TDD strategy reverses the workflow one might expect initially when writing code. This style of programming demands that one starts writing tests before the actual implementation of the problem of interest. When the tests for the implementation pass, a new test is written and the cycle repeats. These small steps of development allow for systematic testing of the project's source code.

System specifications The model simulations are performed on a Zenbook Pro 15 running Windows 10. The most important system specifications are summarized in table 2.

Table 2: Some system specifications of the used Zenbook Pro 15.

System model	ZenBook Pro 15 UX550GEX_UX580GE
Processor	Intel(R) Hexa-Core(TM) i9-8950HK CPU @ 2.90GHz
Installed RAM	16,0 GB (15,9 GB usable)
System type	64-bit operating system, x64-based processor

2.1 Base Case

The setup of the different test cases in this study is constructed such that all individual test cases derive from a base case. A schematic of this base case is presented in fig. 7. The base case describes isothermal uniform flow, in a homogeneous unsaturated soil in the vertical dimension only. The depth of the soil with respect to the soil surface is taken positive in the downward direction and soil type B13, as defined in the Staringreeks [Wösten et al., 2001], is used as default. At the bottom of the system, a system independent boundary condition of type Dirichlet, a fixed state that represent the top of the phreatic surface, is implemented. At the top or soil surface, a system independent boundary condition of type Neumann is implemented, i.e. a fixed flux. *rTop* contains the value of this fixed flux. The nodal discretization of the base case is described by the parameters *N* and *Power* which hold information about the number of nodes and its distribution, respectively. The *biasedspacing* function is used to calculate the nodal discretization, see the *Application Programming Interface (API) Documentation* for the exact definition and additional arguments.

All parameters mentioned in table 3 to table 6 will be varied in the various test cases. In the individual test cases a distinction between two groups of parameters will be made. There are case specific parameters that will be constant but specific for the particular test case and there will be case variable parameters that also differ from the settings in the base case but will be varied during the case calculations. There is a complementary setting for almost every parameter in the *Flow* model in the *Hydrus* model, see *Appendix - Argument map* for a mapping between the parameters that are varied in this study. Note that settings not explicitly mentioned are set to their defaults.

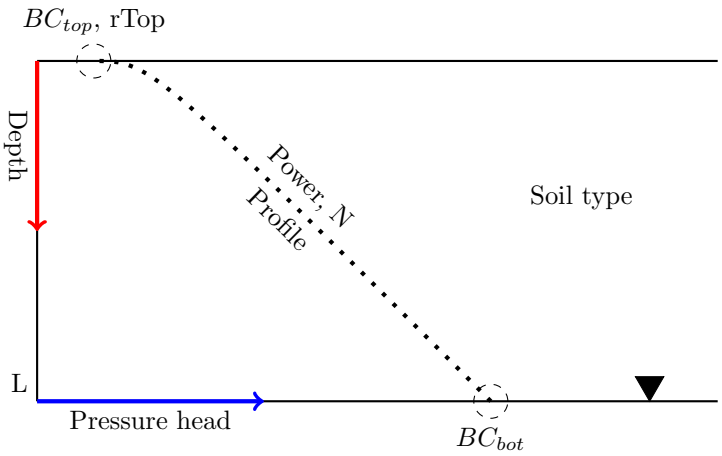


Figure 7: Schematic representation of the pressure head distribution in the base case for vertical flow in the unsaturated zone.

Table 3: Time settings including time step size (dt) and minimum ($dtMin$) and maximum ($dtMax$) allowed time step sizes.

Hydrus / Flow	dt	dtMin	dtMax
unit	*days*	*days*	*days*
value	0.001	0.00001	0.5

Table 4: Convergence settings including the minimum ($ItMin$) and maximum ($ItMax$) number of iterations that decide on increasing ($DMUL$) or decreasing ($DMUL2$) the size of the time step with a particular factor. Also, maximum allowed inner loop iterations ($MaxIt$), total period of time over which the model will be solved ($tMax$) and a conversion criteria ($threshold$) is included.

Hydrus / Flow	ItMin	ItMax	DMul	DMUl2	MaxIt	tMax	threshold
unit	#	#			#	*days*	*cm*
value	3	7	1.5	0.5	500	1	0.001

Table 5: Discretization settings where the length of the system (L), the number of nodes (N) and distribution of nodes ($Power$) are included, see *Application Programming Interface (API) Documentation* for an exact definition of the latter parameter.

Hydrus / Flow	L	N	Power
unit	*cm*	#	#
value	100	101	1

Table 6: Input settings where initial states (s_{init}), a constant flux at the top of the system ($rTop$) and the boundary conditions at both extremes (BC_{bot} & BC_{top}) of the domain are included where the soil type is defined in [Wösten et al., 2001].

Hydrus / Flow	s_{init}	rTop	BC_{bot}	BC_{top}	Soil type
unit	*cm*	*cm/day*			
value	0	-0.1	Dirichlet	Neumann	B13

2.1.1 Cases' source code

Execution of the test cases is performed using Python scripts. To facilitate a structured test suite simple wrappers around the existing models were created to allow for model parametrization using one single settings file. These models were set up such that they could be altered to match the case under consideration. The code presented in the appendix under *Cases' source code* contains the *Flow*, *Hydrus* and *HydrusPicard* wrappers used to conduct analysis for the second case. The latter model mentioned will be discussed in more detail there. It was decided not to include intermediary code such as scripts used for data formatting and plotting.

2.1.2 Case 1 - Analytical Solution

The first test case focuses on solving for the maximum soil depth at which a specific capillary rise flux can be maintained at the ground level. The soil depth is defined as the unsaturated layer between the ground level and the top of the capillary fringe above the water table. The calculation of the maximum soil depth is compared with an analytical solution as given by [Malik et al., 1989]. The maximum soil depths are calculated for a range of capillary fluxes in order to compare the *Flow* model over a large range of suction heads. The distribution of the points at which both models are compared is chosen randomly but within a measurable lower bound of 0.01 cm/day and upper bound of $ksat$ which represent the maximum conductivity of the soil. The minimum allowed time step is set equal to the default of the initial time step dt because of the stationary nature of this model solution and to limit the time it takes for the model to converge. Due to the fact that the gradients of the pressure heads are expected to be large, an increased number of nodes in chosen. The experiment is conducted for three different soils: sand, clay and loam. Parametrization of these soils is based on the soils described by [Wösten et al., 2001]. See table 7 for an overview of the specific and variable parameters of this test case.

Analytical solution as described by Malik The Richards equation, [Richards, 1931], can be used to describe steady-state vertical flow in unsaturated homogeneous soils, see eq. (43).

$$q = K(h)\left(\frac{\partial h}{\partial z} - 1\right) \tag{43}$$

The flux q is calculated as a function of the unsaturated hydraulic conductivity $K(h)$ and the pressure head gradient $\frac{\partial h}{\partial z}$. $K(h)$ is a highly non-linear function which is soil specific. [Gardner, 1958], [R. H. Brooks & A. T. Corey, 1964], [van Genuchten, 1980], [Brandyk and Wesseling, 1985] and others described an unsaturated hydraulic conductivity function which could be used to quantify the capillary rise flux. The analytical solution of [Malik et al., 1989] is used here to compare to *Flow* because his paper proves the validity of this unsaturated hydraulic conductivity function for different soil types. This function is described in eq. (44).

$$k(h) = ae^{-b(h-h_a)} \tag{44}$$

and is defined for $h \geq h_a$ where h_a is the suction at air entry at the top of the capillary fringe, a and b are constants. $k(h)$ is equal to a when $h < h_a$. Substitution of the unsaturated hydraulic conductivity function $k(h)$ in the Richards equation [Richards, 1931], rearranging and taking the integral from zero to infinity leads to the following analytical solution that describes the maximum soil depth for a given capillary flux [Malik et al., 1989]:

Table 7: Case 1 - Settings where the case specific parameters describe the smallest allowed time step size ($dtMin$) and number of nodes (N). The variable parameters in this case are the length or depth (L) and constant boundary flux at the top ($rTop$) of the system including three different soil types which are explicitly defined in [Wösten et al., 2001].

Case specific		Variable		
dtMin	N	L	rTop	Soiltype
days	–	*cm*	*cm/day*	–
0.001	201	$0 - \infty$	-0.01 − -*ksat*	B5-B10-B13

$$z = \frac{1}{b} \ln(1 + \frac{A}{q_m}) \tag{45}$$

where q_m represents the maximum capillary rise flux and $A = ae^{bh_a}$. A can be determined by fitting to experimental data or by making use of the empirical relations described by [Malik et al., 1989] that relate the easily determinable saturated hydraulic conductivity k_s and the moisture content θ_{wp} to the fitting parameters a, b. The pressure head at air entry or h_a is a calculated from these fitting parameters. To relate the purely mathematical model to a realistic situation which represents a soil category, parametrization of the fitting parameters of the exponential unsaturated hydraulic conductivity function $k(h)$ is performed using the empirical relations. The data that characterizes these soils is taken from [Wösten et al., 2001], to calculate the wilting point θ_{wp}, the soil water retention curve as described by [van Genuchten, 1980] is evaluated at a pressure head value of -16000 cm. See table 8 for the parametrization of the exponential unsaturated hydraulic conductivity function $k(h)$. This parametrized unsaturated conductivity function is used in both the analytical solution and the *Flow* model.

Table 8: Case 1 - Fitting parameters calculated by the empirical relation that relates saturated hydraulic conductivity $ksat$ and wilting point θ_{wp} to the fitting parameters a, b and h_a of the exponential unsaturated hydraulic conductivity function described by [Malik et al., 1989].

Category	a cm/day	b day^{-1}	ha cm
Sand	52.91	0.448	27.5
Clay	0.7	0.034	12.2
Loam	12.98	0.166	33.7

2.1.3 Case 2 - Numerical Schemes

In this second test case the numerical schemes of *Flow* and *Hydrus* will be compared. The numerical schemes implemented in these models are the Newton-Raphson and modified Picard iteration schemes, respectively. These iteration schemes are also referred to as inner loop iterations as opposed to outer loop iterations which represent a time step. All inner loop iterations occur within one time step. Unfortunately, inner loop model states are not available to users of *Hydrus*. However, the derivation of the modified picard scheme, the implementation of the Richards equation in *Hydrus* [Šimůnek et al., 2013], is described extensively in literature [Celia et al., 1990] and further discussed and elaborated upon in previous sections of this text. It was decided to replicate the scheme in a new rudimentary model, *HydrusPicard*, written in Python, for the sole purpose of scheme comparison. The elaborated numerical scheme and corresponding script file are included in appendix *Appendix - Hydrus schemes* and *Cases' source code* respectively. The derivation of the scheme for the pressure head based Richards' equation is included for comparison.

The test case consists of multiple parts: To start, the *Flow* model and the *HydrusPicard* model will be compared to *Hydrus*. The models are used to calculate a pressure head profile over depth, where *Hydrus* is used as a benchmark. The pressure head profiles are of transient nature and the profiles will be calculated for several time step sizes. To ensure that the simulation is executed over one time step only, $tMax$, $dtMin$, and $dtMax$ will be equated to the selected time step dt. The other default parameters, such as $ItMin$ and $ItMax$ are not expected to interfere with this model setup because these decision variables are used in outer loop iterations only or $MaxIt$ which defaults to a condition that is not expected to affect the calculation. The comparison will be based on four simulations: a drying and a wetting soil with two different discretizations of equidistant nodal spacing. The drying and wetting simulations will be treated separately and are implemented by changing the top boundary condition BC_{top} to lower and higher pressure head values with respect to the initial states s_{init}. The pressure head profiles are used to compare performance with respect to *Hydrus* and to allow for further comparison. See table 9 for an overview of all case specific and variable parameters used to parametrize this test case.

Table 9: Case 2 - Settings concerning the case specific parameters include the bottom boundary condition (BC_{bot}) of type Dirichlet and initial states (s_{init}) which are defined by the linear interpolation between the bottom boundary state and -100 cm pressure head at the soil surface by a number of evenly spaced (N) nodes. Other case variable parameters are the Dirichlet boundary condition at the top (BC_{top}) of the system, several time related parameters describing the current time step size (dt), minimum ($dtMin$) and maximum ($dtMax$) time step sizes and the complete simulation time ($tMax$). Gaussian quadrature procedure is quantified and varied by changing degrees (Λ) of integration. The total number of simulations per model, excluding (*), is the product of the variable parameters ($2 * 2 * 4$). Note that the elements indicated (*) are used for the Gaussian quadrature simulation only.

Case specific		Variable			
BC_{bot}	s_{init}	N	BC_{top}	$dt/tMax/dtMin/dtMax$	Λ*
cm	cm	$-$	cm	min	-
-10	linspace	(101, 51)	(-300, -10)	(1, 2, 10, 60, 100*)	1-5

Secondly, the convergence behavior of *Flow* and *HydrusPicard* will be examined and compared for all individual situations as discussed in the first part. Convergence for both models will be quantified on a per iteration basis. This is calculated by taking the difference or absolute difference of the states at the current iteration and its stable solution. These differences of the states at the nodes are aggregated and normalized to result in a single value. The result is used to examine how the iteration schemes under consideration progress towards its solution and whether it oscillates. Quantifying convergence per iteration is calculated using

$$\frac{\Sigma_i^N (s_i - s_{i,stable})}{N} \text{ or } \frac{\Sigma_i^N |s_i - s_{i,stable}|}{N} \tag{46}$$

where the definitions of the symbols are as used earlier, which are also listed in *Glossary*. Additionally, convergence will be discussed on the basis of iterations needed to reach a stable solution.

Finally, the impact of changing the Gaussian quadrature degree of *Flow* is examined in terms of iterations needed for conversion and compared to the solution of the *Hydrus* model. The *HydrusPicard* model will not be presented here because the point of interest is shifted to conversion on an iteration basis for which the rudimentary *HydrusPicard* model does not add any more value than *Hydrus* itself, i.e. profile data from inner loop iterations is not used for analysis here. In this last part of the test case only the wetting situation is considered because convergence behavior is expected to become more critical in these situations due to high pressure head gradients, [Celia et al., 1990].

2.1.4 Case 3 - Time Complexity

In this experiment the time complexity of *Hydrus* and *Flow* will be examined and compared, both qualitatively and quantitatively. Here, the time complexity is defined as the time the computer needs to complete a specific model simulation. The complete experiment will be repeated ten times in order to correct for the random noise among the results of the repeated experiment, see table 10 for the case specific and variable parameters.

Fig. 17 in the appendix shows the complete schematic solve procedure of the *Flow* model. The 'variable time block', which is selected with the dashed ellipse, influences the time complexity of the model the most because it determines the time step size dt for the Newton-Rapshon iteration [Ypma, 1995]. This experiment is focussed on the 'variable time block' taking into account that the evolution of the time step size is not limited by its boundary conditions in order to measure an undisturbed dependency on its variable parameters $DMul$ and $Dmul2$.

The 'variable time block' is presented in fig. 4. Detailed representations of 'dt_solve' and 'dt_block' are presented in fig. 5. These flowcharts were discussed in the previous section *Transience of the System*. The case specific parameters are chosen such that the 'variable time block' is isolated from the remaining part of the complete solve procedure. This means that the 'variable time block' procedure should only terminate on $tMax$ being exceeded by the current time t. To achieve this behavior the number of inner loop or Newton-Raphson iterations IT should not exceed the the maximum number of iterations allowed $MaxIt$. Additionally, the size of the time step dt should not be limited by any lower $dtMin$ of upper $dtMax$ bound. The constraints are formalized in table 11.

Table 10: Case 3 - Settings where the case specific parameters include the time step size (dt), maximum allowed time step size ($dtMax$), maximum number of allowed iterations per time step ($MaxIt$) and the value of the constant boundary flux ($rTop$) at the top of the system. The case variable parameters include two multipliers to increase ($DMUL$) or decrease ($DMUL2$) the time step size. The multiplier ranges are varied by taking 15 equally spaced values including the boundaries.

Case specific					Variable	
dt	dtMax	MaxIt	tMax	rTop	DMul	DMul2
days	*days*	-	*days*	*cm/day*	-	-
0.0004	2.0	10	4.0	-0.9	1.2 - 1.9	0.2 - 0.9

Table 11: Case 3 - Constraints that are not desired to affect the time step (dt) evolution at the lower ($dtMin$) or upper ($dtMax$) side of the spectrum where number inner loop of iterations (IT) may not exceed the threshold ($MaxIt$) set.

Contraint
$IT < MaxIt$
$dt > dtMin$
$dt < dtMax$

3 Results

3.1 Case 1

The comparison of the maximum capillary rise flux calculated by the analytical solution of [Malik et al., 1989] and *Flow* are presented in fig. 8. The result shows that the behavior of the capillary rise with respect to the different soil types is as might be expected from the analytical solution [Malik et al., 1989]. The gradient of the soil depth with respect to the maximum capillary flux increases from clay to loam to sand which is captured by the fitting parameter b. The highest capillary flux a soil can maintain, which is the saturated hydraulic conductivity by definition, is equal to the suction at air entry h_a at the top of the capillary fringe. This means that the unsaturated length of the soil at these positions approach zero which is in agreement with eq. (45). See table 8 for the exact values of the fitting parameters. At the other extreme of the spectrum, where the maximum capillary rise flux approaches low values, the graph theoretically extends to infinite soil depths. The value of 0.01 cm/day for the capillary rise flux is set to be the lower bound of the capillary flux spectrum which value is in the order of the measurement error. All three soils can maintain this flux at the soil surface in sand, loam and clay ordered for increasing soil depths. This relation was also found in other literature [Malik et al., 1989], [Wösten et al., 2001].

Due to the logarithmic scale in fig. 8 deviations are hard to notice visually. In table 12 the differences between the analytical solution and the *Flow* model are presented using a statistical measure (RMSE). These results show deviations in the order of 10^{-4} - 10^{-5} centimeters. The distribution of these deviations do not show a significant trend in either direction. All deviations are positive which mean that the analytically calculated values all show slightly larger values. All deviations have lower magnitude than the conversion threshold described in table 4 and the deviations in the different soil categories are distributed randomly, see fig. 16 in the appendix.

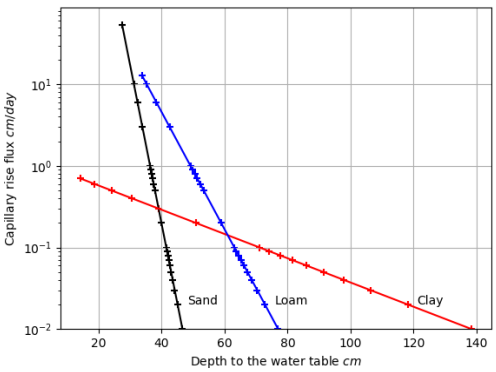


Figure 8: Case 1 - Comparison of *Flow* (dots) with the analytical solution (lines) of [Malik et al., 1989] for 3 different soil types.

Table 12: Case 1 - Root mean square error (RMSE) between the analytical solution of [Malik et al., 1989] and the *Flow* model.

Category	RMSE (cm)
Sand	1.03e-04
Clay	3.98e-05
Loam	7.74e-05

3.2 Case 2

Pressure head profiles as a function of depth are presented in fig. 9 for qualitative observations of the simulated profiles. In table 13 a quantitative measure, root mean square error (RMSE), for each simulation with respect to the corresponding *Hydrus* solution is presented.

Pressure head profiles for the drying situation at $N = 101$ are presented in the upper two left panels of fig. 9. Visual assessment of the graph shows clear differences. Both models perform differently with respect to their benchmark. A better fit is found for the *Flow* models as opposed to *HydrusPicard*. Additionally, model fits for *HydrusPicard* seem to overestimate the solution found by the referencing *Hydrus* solution, i.e. the solution profile is ahead of its benchmark. Referring to table 13 quantifies above subjective qualifications. From this table is read that errors of *Flow* with respect to its benchmark are smaller than corresponding solutions of *HydrusPicard* at the same time step with its benchmark. In addition, both models show an error divergence as function of increasing time step size where this relation is found to be stronger for *HydrusPicard*, e.g. 0.16 and 0.87 cm as difference of the deviations between the extreme time steps.

Shifting the focus to the lower left two panels where pressure head profiles for the drying situation at $N = 51$ are shown, one clear difference with respect to the previous situations catches the eye. By visual inspection of fig. 9 errors of *Flow* with respect to its benchmark are noticed, especially for the larger two time steps. *Flow* underestimates the benchmark solution, i.e. the profile lags behind. Confirmation of the observed differences is presented in table 13. This table also shows that the model solutions of *HydrusPicard* more closely match to the benchmark than the *Flow* model for the smaller two time steps. For the larger two time steps the opposite is true. The relationship of increasing deviation as a function of increasing time step is found for both models where *HydrusPicard* shows the strongest relationship, in numbers expressed as 0.3 and 0.83 cm calculated as the difference between the deviations at the extremes of the time steps.

The main differences found between the model solutions taking the change in discretization into account for the drying situation are the underestimation and overestimation of *Flow* and *HydrusPicard* solutions compared with their benchmarks, respectively. Additionally, deviations of *Flow* solutions show comparable performances especially at smaller time step sizes, e.g. errors of *Flow* lie between 0.01 and 0.02 cm, where error differences are larger for the largest two time steps (0.04 and 0.16 cm). This variation is less clear in the case of the *HydrusPicard* model. The smallest difference in deviation is 0.06 cm at the smallest three time steps, at the largest time step $dt = 60$ the error deviation is found to be 0.09 cm.

In the upper two right panels of fig. 9 pressure head profiles of a wetting situation are presented where $N = 101$. Visual qualification shows that *Flow* calculates an underestimation of the profile for the smaller two time steps while *HydrusPicard* presents solutions that overestimate

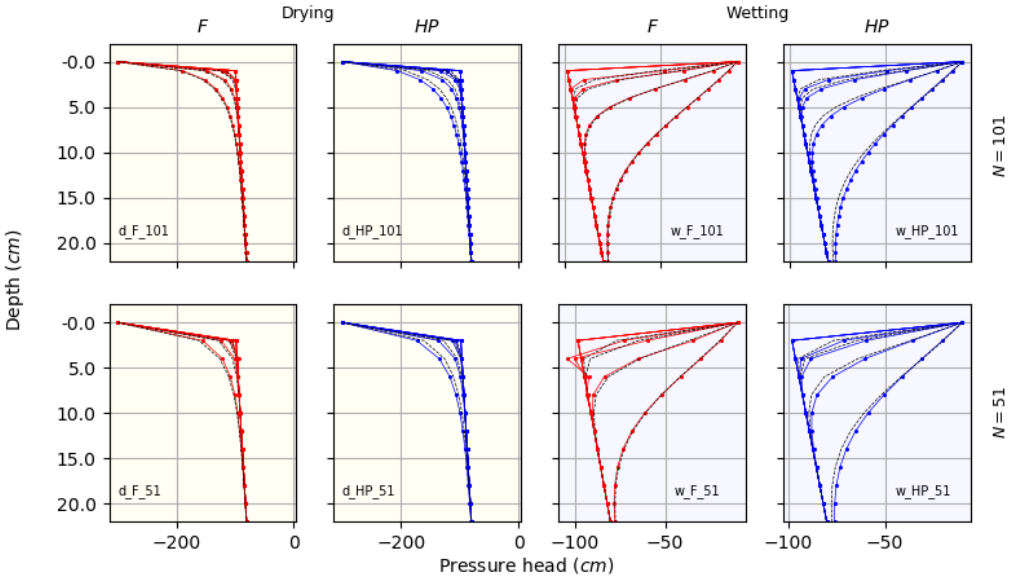


Figure 9: Case 2 - Pressure head depth profiles for drying(d)(yellow)/wetting(w)(blue) curves for *Flow (F)* (red lines) and *HydrusPicard (HP)* (blue lines) at two different ($N = 101, N = 51$) equidistant nodal spacings. Note that the plots are zoomed to the region of interest. Each individual plot contains simulations where increasing time steps dt are used, see table 9. For all simulations a *Hydrus* (black dashed lines) benchmark simulation is plotted.

the benchmark for all time steps. In table 13 is shown that errors for the smaller two time steps are indeed larger for the *Flow* model. Whether the profile also underestimates at the last two time step sizes can not be concluded from fig. 9 or table 13. Additionally, the change of deviations in the latter table mentioned for *Flow* is found to be negatively related to increasing time step sizes while the same relation for *HydrusPicard* is found to be positively related. The above statement in numbers reads |-0.06| and 0.31 cm for the difference of deviations at the extreme time steps where the strongest trend is found for the *HydrusPicard* model.

The last situation to be discussed is presented in the lower two right panels of fig. 9. In this wetting situation discretization coarsens to $N = 51$. What stands out from visual inspection is the oscillations found for the *Flow* model for the two smallest time steps. The *HydrusPicard* model does not show any oscillations. In addition, model solutions of *Flow* tend to underestimate its benchmark which cannot be verified for the largest time step. The model solutions of *HydrusPicard* overestimate the benchmark for all time step sizes. These observations are supported by data in table 13. Deviations with respect to the benchmarks are negatively (|-0.22| cm) and positively (0.3 cm) related to the increase of time step size for *Flow* and *HydrusPicard* respectively where the strongest trend is found for the *HydrusPicard* model.

Differences for the wetting situation that originate from difference in discretization show the *Flow* model to be more sensitive to coarsening of discretization than the *HydrusPicard* model, especially at smaller time steps. Underestimation of *Flow* and overestimation of *HydrusPicard* occurs for both discretizations. The magnitude of the difference of deviations in the over- and underestimations within the models itself are larger at $N = 51$ especially for the *Flow* model and smaller time steps, e.g. at $dt = 1$ the difference in deviations is 0.2 and 0.03 cm, and at $dt = 60$

the difference in deviations is 0.05 and 0.02 cm for *Flow* and *HydrusPicard* respectively.

Comparing both situations, drying and wetting, the following general trends can be observed. To begin, models correspond more closely to the benchmark simulation where nodal density is highest ($N = 101$), except for *HydrusPicard* in the drying situation. Secondly, in the drying situation all deviations with respect to the benchmarks increase with increasing time step size where the relation is strongest for *HydrusPicard*. In the wetting situation the relation is reversed for the *Flow* model. The relations are still strongest for the *HydrusPicard* model, however less distinct in the wetting case at $N = 51$. Finally, *Flow* shows oscillations in the solution of the wetting situation, which is not observed in the drying simulations or the *HydrusPicard* model at all.

In fig. 10 and fig. 11, drying and wetting situations are presented separately, convergence behavior of *Flow* and *HydrusPicard* are presented for all time steps and both discretizations on a per iteration basis using two different statistical measures as defined by eq. (46) called C_{sum} and C_{abssum} respectively. The convergence behavior of both drying and wetting situations is summarized and combined in fig. 12 on the basis of total iteration number.

Convergence behavior of the models for the drying situation at $N = 101$ are displayed in the top panels of fig. 10. For *Flow* C_{sum} is positive after the first iteration for all time steps, where larger time step sizes lead to increasing convergence sums after the first iteration. However, the larger the conversion sum after the first iteration, the faster it converges as function of iteration number. The trajectory of convergence is positive over the whole domain, in other words, the normalized scalar convergence sum does not oscillate such that the center of mass becomes negative. The center of mass of a particular solution is viewed as the weighed over- and underestimation of the benchmark averaged over all nodes in the domain.

The *HydrusPicard* model in the top mid panel starts off with positive values for C_{sum} where relation between time step size and value of the statistic is shown to be positive at the first iteration number. Convergence of model runs where time steps are smaller show the fastest convergence. However, convergence sums do not remain positive. C_{sum} oscillates towards stable solutions where the magnitude of oscillations increases for increasing time steps, e.g. -0.9 and -3.4 cm for $dt = 10$ and $dt = 60$ respectively after two iterations.

In the top right panel the normalized scalar absolute conversion sum, C_{abssum}, is plotted against iteration number. It is clear that the *Flow* model converges more quickly than the *HydrusPicard* model as function of iteration number where convergence of *Flow* appears exponential, linear on a log scale, while convergence of *HydrusPicard* appears to be linear or of lower

Table 13: Case 2 - Root mean square error (RMSE) with the corresponding *Hydrus* solution in *cm* for all situations and four different time steps. The index column is read as drying(d)/wetting(w)_*Flow*(F)/*HydrusPicard*(HP)_*N(101/51)*.

simulation / *dt*	1 min	2 min	10 min	60 min
d_F_101	0.05	0.08	0.15	0.21
d_HP_101	0.09	0.14	0.38	0.96
d_F_51	0.07	0.09	0.19	0.37
d_HP_51	0.04	0.08	0.32	0.87
w_F_101	0.13	0.10	0.05	0.07
w_HP_101	0.11	0.15	0.27	0.42
w_F_51	0.33	0.27	0.16	0.11
w_HP_51	0.14	0.17	0.29	0.44

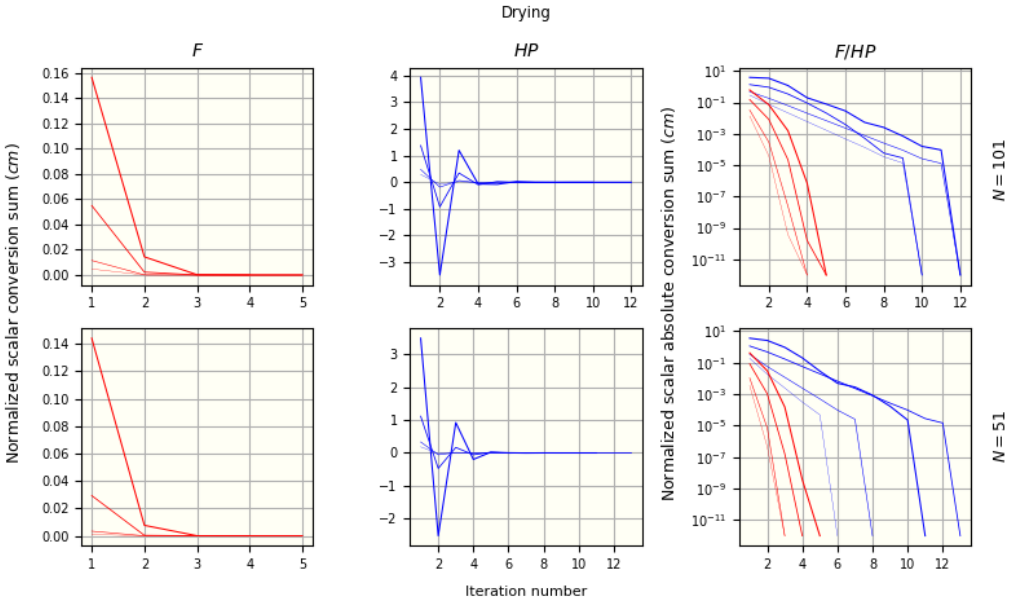


Figure 10: Case 2 - Convergence behavior of *Flow (F)*(red lines) and *HydrusPicard (HP)*(blue lines) as a function of iteration number for the drying situation separated by nodal density N. Conversion is calculated as the difference or absolute difference of the values at the nodes at a specific iteration and the final stable profile at $t = dt$. This result is aggregated and normalized to a scalar value. Increasing line thickness corresponds with increasing time step sizes. The conversion formula and time step sizes used are given by eq. (46) and in table 9 respectively.

exponential power. Additionally, the number of iterations needed for *Flow* to converge is larger or equal to the iterations needed for the previous time step simulation to converge while this is not the case for the *HydrusPicard* model, e.g. iteration number (12) at convergence for $dt = 2$ is larger than the iteration number (10) of convergence at $dt = 10$.

Shifting discussion of the drying situation to the lower three panels where $N = 51$, few differences with the above three panels are observed. The convergence behavior of both *Flow* and *Hydrus* are similar, where C_{sum} values are of the same order of magnitude. Focussing on the bottom right panel small differences are observed. The *Flow* model needs one iteration less for the lower three time steps, convergence behavior in terms of iterations for the largest time step remains the same. Larger differences with respect to iterations needed for conversion are observed for *Hydrus*, especially for the smaller two time steps, e.g. iteration number shifts from 10 and 12 to 6 and 8 for the smaller two time steps respectively, while iteration numbers of both the largest two time steps only increase by 1. Values for the conversion statistic C_{abssum} do not show significant differences with respect to its counterpart.

Convergence behavior of the models simulated for the wetting situations are presented in fig. 11 where the top three panels contain simulations at $N = 101$. The normalized scalar conversion sum C_{sum} of the *Flow* simulations is positive after the first iteration for all time steps, but the center of mass does not remain positive as function of iteration number. Progression towards the stable solution shows oscillations where the magnitude of these oscillations increase for increasing time steps.

The values of C_{sum} decrease with increasing time step after first iteration for the *HydrusPicard*

simulations, shown in the top mid panel. The center of mass is predominantly negative over the whole solution domain, where a small fluctuation above the zero line can be observed at iteration number 5 for the largest time step.

The top right panel shows comparable results in terms of iteration number, where *Flow* is on the lower end (6 and 7) while *HydrusPicard* is on the upper end (11 and 12). The differences between the order of convergence in terms of C_{abssum} is less clear. While convergence of *HydrusPicard* remains loglinear, convergence of *Flow* appears to be a combination of linear and exponential convergence. Note that C_{abssum} values are plotted on a log scale. In addition, while for *HydrusPicard* a smaller time step results in a faster convergence in terms of iterations, the *Flow* model does not obey to this relation, e.g. the iteration number at convergence for $dt = 2$ is larger than the corresponding value at $t = 10$.

The last simulations discussed, wetting and at $N = 51$, are shown in the lower three panels. The order of magnitude of C_{sum} is of the same order as its $N = 101$ counterpart comparing between the models. What stands out for the *Flow* model is that the magnitude of oscillation for the largest time step is dampened out less slowly. A small single period oscillation of the C_{sum} statistic for the *HydrusPicard* model can be observed for the largest two time steps at iteration number 3 to 5.

Convergence behavior in terms of C_{abssum} is shown in the bottom right panel. The positive relation, as earlier described, of iteration number needed for the model to converge as function of time step size holds for both models. Also, the order of convergence is comparable to the results for $N = 101$. In other words, all *HydrusPicard* models converge loglinearly, while convergence for *Flow* appears to be exponential, especially for the two smallest time steps. Convergence for the larger two time steps is a combination of linear and exponential convergence.

Iteration data from all right panels in fig. 10 and fig. 11 is summarized in fig. 12 where the results from the *Hydrus* benchmark runs are added. Some general statements about the results can be deduced from the data in this form more easily.

All *Flow* simulations perform better in terms of iteration number with respect to the corresponding *HydrusPicard* model except for the wetting situation at both discretizations for time step size $dt = 2$ and $dt = 60$. For *Flow*, iteration numbers are always smaller for the drying situation while the same comparison for *HydrusPicard* appears to be more randomly, e.g. the highest number of iterations is observed for the drying situation at $dt = 10$ while the lowest iteration number is 6 for both wetting and drying model runs. Sensitivity, in terms of iteration number, to discretization is smallest for the *Flow* model in the drying situation where the maximum difference is at most one single iteration less in case of the coarsest discretization $N = 51$. The strongest dependencies on discretization are also observed for the *Flow* model at the wetting situation. Dependencies on discretization for the *HydrusPicard* model do not show a clear difference between drying and wetting situations, but it can be noted that differences tend to decrease with increasing time step size dt.

Performance of the *Hydrus* model in terms of iteration number compares more closely to *Flow* than to *HydrusPicard*. First of all, the average iteration number across situation and discretization is lowest for *Hydrus* at all time steps. In addition to this, *Hydrus* is observed to show the smallest difference in iteration number between the drying and wetting situations where these difference even shrink for increasing time step sizes. The latter observation also holds for the differences in nodal discretizations where the only exception is the *Flow* model for the drying situation at the two smallest time steps.

In fig. 13 performance of the *Flow* model as function of changing Gaussian quadrature degree (Λ) compared to the *Hydrus* model is shown. The first two bars at each time step are the same model simulations as performed and presented above for the wetting simulation, see fig. 11, where the default value $\lambda = 1$ was set for *Flow*.

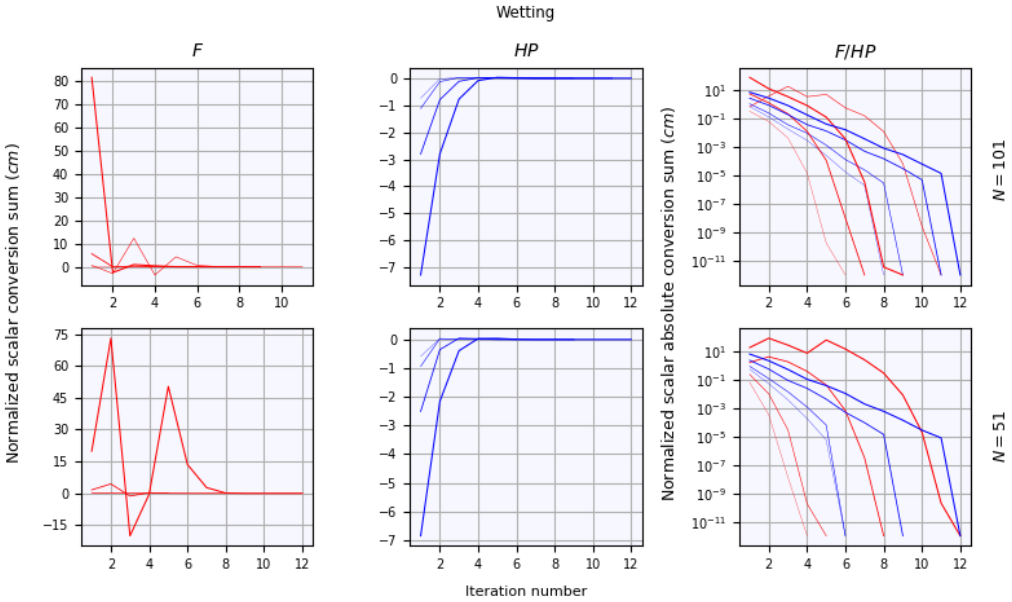


Figure 11: Case 2 - Convergence behavior of *Flow (F)*(red lines) and *HydrusPicard (HP)*(blue lines) as a function of iteration number for the wetting situation separated by nodal density N. Conversion is calculated as the difference or absolute difference of the values at the nodes at a specific iteration and the final stable profile at $t = dt$. The result is aggregated and normalized to a scalar value. Increasing line thickness corresponds with increasing time steps sizes. The conversion formula and time step sizes used are given by eq. (46) and in table 9 respectively.

Focussing on the top panel, a strong response in iterations needed for convergence is observed when the Gaussian quadrature degree is incremented from 1 to 2. Further increases of λ do not have this clear effect. The number of iterations needed for convergence appears to vary randomly or remain constant for higher orders of λ. At these higher Gaussian quadrature degrees, time step size does not affect the number of iterations needed for convergence in any predictable manner; results remain the same or show little variation for the largest two time steps. Concerning the root mean square errors (RMSE), differences with respect to the *Hydrus* solution show smallest errors for the smaller time steps and these errors increase as function of time step size dt. This relation is inverted as function of higher Gaussian quadrature degree. Errors with respect to the benchmark are largest for the smallest time steps and decrease as function of increasing time step.

In the lower panel where discretization is coarser, the first observation made is the absence of a *Flow* solution for all Gaussian quadrature degrees larger than 1 at the smallest time step $dt = 1$. For the next time step where $dt = 2$, the highest number of iterations needed for convergence is found (19). No variation in iteration number is found for $\lambda > 1$ for all other time steps except for $dt = 60$ where small variations are observed. Futhermore, strong sensitivity to λ between degree 1 and 2 is still observed, also the behavior of the root mean square error (RMSE) is observed to be similar.

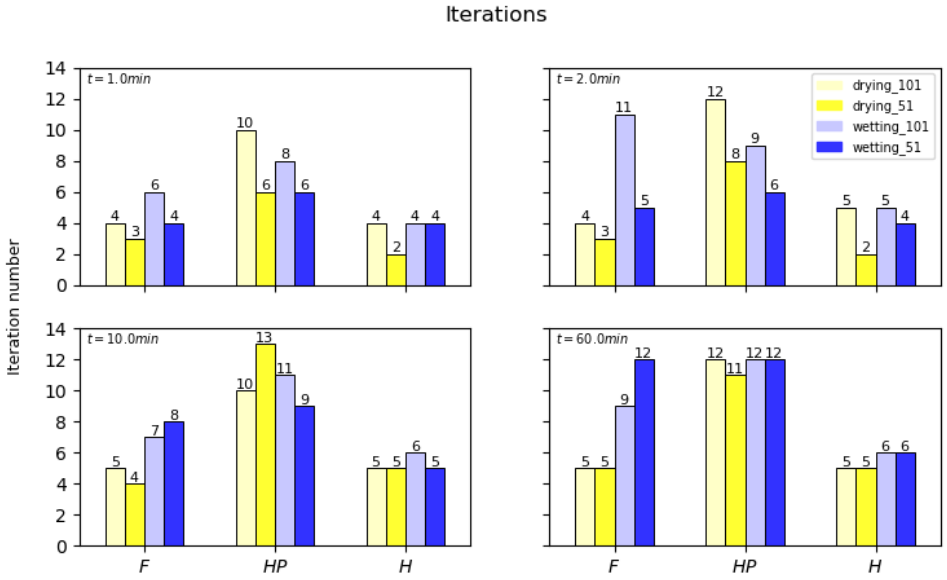


Figure 12: Case 2 - Number of iterations needed for *Flow (F)*, *HydrusPicard (HP)* and *Hydrus (H)* to converge to a stable solution at several time step sizes dt for four different situations labeled as $drying/wetting\_N(101/51)$.

3.3 Case 3

The first results of conducting the experiment described in case 3 are presented in fig. 14. What stands out immediately from these results is that the order of magnitude of the run times differ approximately with a factor of 100 between the models *Hydrus* and *Flow*. However, the patterns of the run times as a function of the variable parameters $Dmul2$ and $DMul$ seem to be in agreement. The plots of the run times of both models show a decrease as a function of increasing $DMul$ and do not show a strong dependence on $DMul2$. In addition, the highest absolute run time of both models is found at (1.2, 0.2) for $DMul2$ and $DMul$ respectively.

The constraints imposed on the model runs are not exceeded which means that the time step dt was not limited by its boundaries $dtMin$ and $dtMax$. It also means that the number of iterations IT was not limited by $MaxIt$. Therefore, all the solution finding procedures terminated on the current time t being equal to the end time $tMax$ decision block as shown in the 'variable time block' in fig. 17. The constraints not being exceeded is proven in fig. 18 in the appendix. The right part, fig. 18b, shows another interesting characteristic of the model runs. The evolution of the number of iterations exceeds the lower boundary $ItMin$ but exceeds the upper boundary $ItMax$ in a much lesser degree. The latter only occurs at some of the first time steps when the change in states is highest. This observation suggests that the model run time solution spaces in fig. 14b do not show a dependency on $DMul2$ because there is none, but it simply shows no dependency because of the fact that in almost none of the iterations the $ItMax$ limit was reached and therefore the time step size was not lowered as a consequence of this. However, this limit is exceeded in both models but only at the beginning of the solve procedure. As a consequence, the trend in the horizontal directions of the run time surfaces in fig. 14 are not completely horizontal but a little skewed towards the right. The direction in which the horizontal is being skewed is

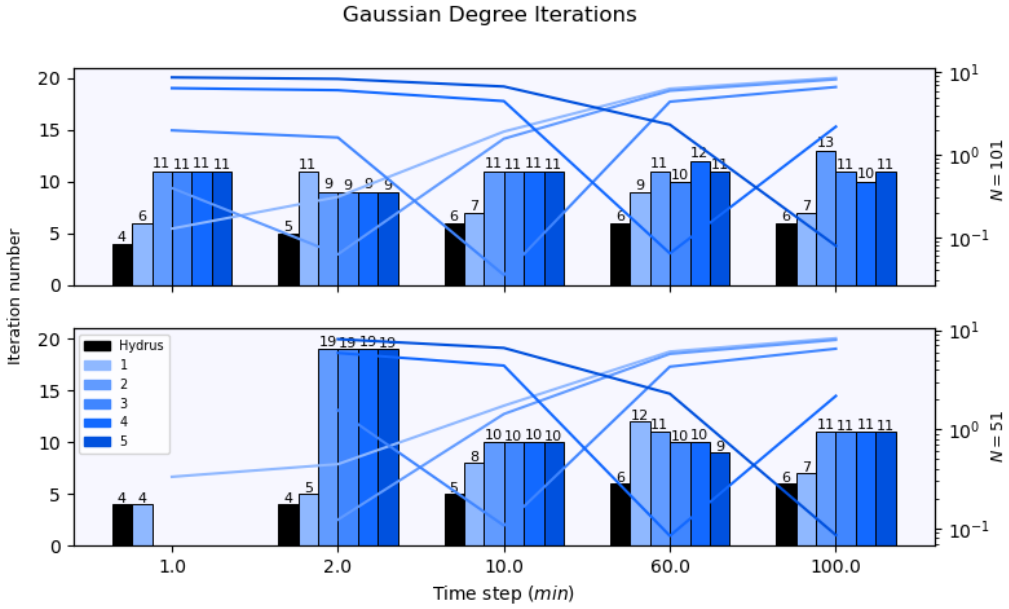
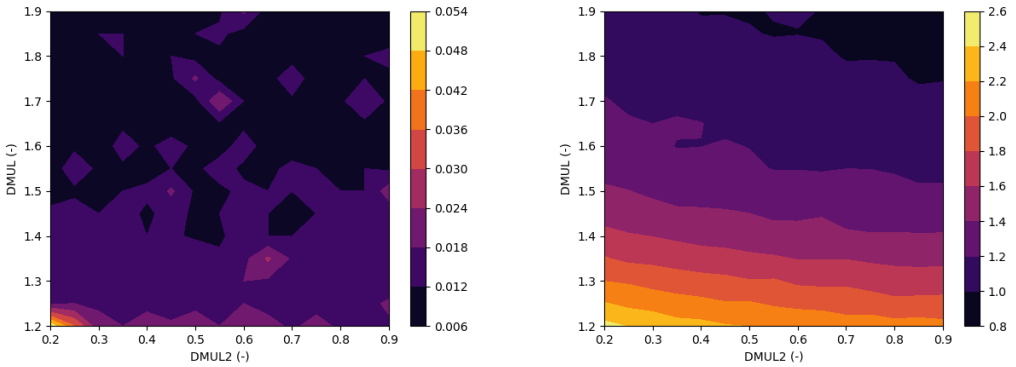


Figure 13: Case 2 - Number of iterations needed for *Hydrus* (black bars) and *Flow* (blue bars) simulations to converge using increasing Gaussian quadrature degrees from 1 to 5 corresponding with increasing color bar saturations. The models are run for 5 separate time step sizes dt and two different nodal discretizations N. On the secondary axis, the root mean square error (cm) with the corresponding *Hydrus* solution as function of time step is presented.

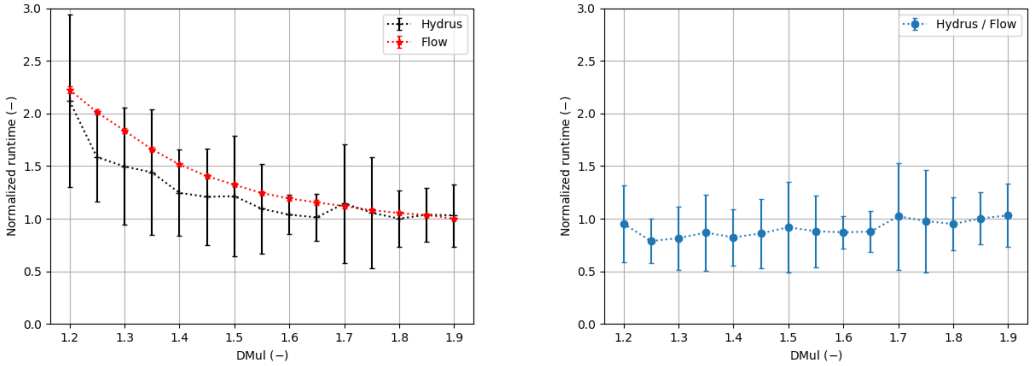
as expected because a larger value of $DMul2$ will result in a smaller time step decrease in the next Newton-Raphson [Ypma, 1995] iteration and therefore $tMax$ is reached in fewer time steps. The lower boundary $ItMin$ influences the time step more often until a fluctuating or stable time step is developed and therefore showing a strong dependency in the $Dmul$ variable parameter. The direction of the trend in the vertical of both run time surfaces from fig. 14 is decreasing as a function of $DMul$ as a result of this parameter having the opposite effect of $DMul2$. Larger values for $DMul$ result in time steps that increase its magnitude quicker as opposed to lower values of this parameter.

In fig. 15 the normalized run times, including the standard deviation calculated from ten separate runs, of both models is presented. The model solutions from fig. 14 were averaged over the weakly dependent variable parameter $DMul2$. As a result, one-dimensional time complexity plots remain. In fig. 15a both models' dependencies are shown as a function of $DMul$, these normalized run times increase for a decreasing explanatory variable. The normalized standard deviation differs significantly, showing much larger variation for the run times in *Hydrus* than for *Flow*. Despite this difference between the normalized standard deviation between the models, the behavior of both models is observed to be equal as a function of $DMul$, see fig. 15b. In this plot the ratio of the run times of the one-dimensional plots is presented. This result shows constant behavior because the ratio of unity and the fluctuation of the ratio around this value being within uncertainty bounds.



(a) Hydrus (b) Flow

Figure 14: Case 3 - Average duration of solving both numerical models as function of $DMUL$ and $DMUL2$ which respectively increase and decrease the time step size during the simulation of evaporation from the top soil for four days. The average durations are obtained from 10 individual runs presented in seconds.



(a) Normalized run time. (b) Normalized run time ratio.

Figure 15: Case 3 - Normalized run times including standard deviation from ten separate runs.

4 Discussion & Conclusion

In this study the performance of the *Flow* model is compared to an analytical model [Malik et al., 1989] applied for a sand, loam and clay soil in order to verify the *Flow* model. Both models do not show different behavior of the soil depth relations with respect to the capillary flux at the soil surface. The sand soil shows the weakest relation where flux ranges are largest while the clay soil shows the strongest relations where the capillary flux range was smallest. Also, behavior at the extremes of the domain was comparable, i.e. capillary fluxes approaching zero indicate infinite soil depths while a flux valued at the saturated hydraulic conductivity approaches infinitely small soil depths. Sampling the capillary flux values randomly and verifying the soil depths to the analytical solution resulted in equally good comparisons independent from soil type where all fitting differences were smaller than the conversion threshold set. As verification succeeded, comparison with the *Hydrus* model was justified.

The center of mass of the particular solution at specific iteration steps is useful for giving an indication of the stability of the solution in terms of oscillation, divergence or convergence with respect to the benchmark. The comparison with this model on the basis of numerical solution scheme showed that for a drying situation the Newton-Raphson iterations converge to the solution at the selected time step such that the center of mass does not become negative. The Picard iterations do show that the center of mass oscillates between positive and negative values which is strongest for larger time step sizes. Here, the *Flow* model converges at least one order faster than the *Hydrus* model, irrespective of order of discretization. For the wetting situation behavior was found to be different. The Newton-Raphson iterations show oscillations of the center of mass as function of iteration number, however this is dampened more strongly for the finest discretization. More importantly, the behavior of the Picard iteration scheme does not show oscillations and the center of mass of the iterative solutions are generally negative. The difference in order of convergence over the different discretizations is comparable between the models where *Flow* shows the strongest iteration number difference on this variation compared to the drying situation. In general, *Flow* simulations perform better in terms of iterations needed for convergence. Especially for the *Flow* model, iterations in the drying situation are less than for the wetting simulation. Also, sensitivity to discretization is largest for this model in the wetting situation. Increasing time step sizes tend to increase the deviations from the benchmark, except for the *Flow* model in the wetting situation where deviations tend to decrease. Where the Gaussian quadrature was changed from one to higher orders for the *Flow* model in the wetting situation, iterations needed for convergence strongly increased but appear to be relatively insensitive to higher orders of the Gaussian quadrature degrees.

Regarding the time complexity, the $DMUL2$ parameter was shown to have a negligible effect on the time complexity due to the case setup. The time complexity was examined as function of $DMUL$ where the run time durations decrease for increasing value of this explanatory variable for both *Flow* and *Hydrus*. This relation is equal between both models because the normalized runtime ratio did not significantly differ from unity. The absolute run time durations differ approximately a factor of 100, where the *Hydrus* model is fastest.

The verification of *Flow* is justified based on the unique relation of capillary flux and soil depth for a specific soil parametrization, see eq. (45). However, the verification of the model is only tested in the form of a final result. Internal evolution of the model solution could not be verified as a consequence of the analytical nature of the function which was compared with. However, the behavior of the evolution of the internal model solution was compared with the *HydrusPicard* model and existing literature. Performance of both the *Flow* and *HydrusPicard* model with respect to the states of the corresponding *Hydrus* solution is of comparable order for the drying

situation. The wetting situation shows significant differences with respect to this measure. It is well known that in the situation of infiltration into dry soils unstable solutions arise where finite element schemes may exhibit oscillatory solutions [Celia et al., 1990]. This oscillation is magnified by a coarser nodal grid but reduced by increase of the time step size. This is in contrast with increase of the time step size in the drying situation where state differences with respect to the benchmark increase in that direction. Although, smaller time step sizes are still preferred [Zha et al., 2019], the sharp infiltration front is more dispersed and therefore less sharp after a longer time interval which would explain the opposite behavior. The *HydrusPicard* model does not show any oscillations but the increase of state differences with respect to the benchmark is consistently found for increasing time step size where coarser nodal grid is of less influence. Negation of the center of mass in internal iterations is dependent on the direction towards the stable solution for the modified picard method while evolution to the stable solution using the Newton-Raphson scheme is mainly positive and independent of direction towards the stable solution. Oscillation with respect to this measure is found for the *Flow* model in the wetting simulation but is caused by the unstable solutions that are found for these model runs. According to literature, the Newton-Raphson scheme converges quadratically [Brutsaert, 1971] while the modified picard iteration scheme converges linearly [List and Radu, 2016]. The different order of convergence was also found in the model simulations where in most situations the *Flow* model needed less iterations for convergence as opposed to the *HydrusPicard* model. The less distinct difference in convergence behavior between both models in the wetting situation is explained by the oscillating stable solution. On a general note concerning model performances on the basis of spatial and iteration schemes it is found that finite differences and finite element implementations do not show significantly different performances. In addition, performances of modified Picard method or Newton-Raphson iteration schemes are not significantly different either [Farthing and Ogden, 2017], [Zha et al., 2019]. What supports this observation even further is that many popular simulation software implements all combinations of the spatial and iterative schemes mentioned. It is important to note that, as shown in this paper, there are situations where one implementation is preferred over the other, but no combination of implementations is found to be consistently better [Zha et al., 2019].

In the last part of the experiment the Gaussian quadrature degree was varied. This essentially affects the locations within a segment where the Richards equations was sampled in order to approach a more realistic flux value over the interval that will be distributed to the nearest neighboring nodes. The strong response found when incrementing the Gaussian quadrature from one to two reflects that the initial situation might not effectively capture the non-linear behavior of the governing flow equation. Increasing the Gaussian quadrature degree further did not show this strong response anymore which might indicate a better representation of the flux over the interval that is distributed to the nodes which is not affected by sampling at smaller intervals anymore. The inverse relation of Gaussian quadrature degree and time step size can be explained by the size of internal pressure head gradients in the system. Small Gaussian quadrature degrees, which mean relatively large nodal spacings, provide better convergence to the benchmark for smaller time step sizes than for larger time step sizes where not enough detail is concerned. However, when Gaussian quadrature degrees increase the internal gradients are too large over the relatively small nodal segments at small time step sizes, therefore solutions converge to the benchmark more closely for larger time steps where the internal gradient is flattened over time.

In the last experiment the time stepping scheme of both *Flow* and *Hydrus* was compared. Results indicated that the internal time stepping scheme between the models was comparable as normalized run time ratios not significantly differ from unity. However, the time it takes for models to find a solution was found to be different as was expected [Merelo-Guervós et al., 2016], [Kwame et al., 2017] as a consequence of implementations in different programming languages

where interpreted languages (e.g. Python, Perl, JavaScript) are generally orders of magnitudes slower than compiled languages (e.g. Fortran, C, C++). The internal time stepping algorithm was not affected by these fundamental implementation differences despite the fact that internal data structures and program design have the potential to significantly affect a program's performance [Merelo-Guervós et al., 2016].

Being more precise on the verification of the *Flow* model with respect to the analytical solution by [Malik et al., 1989] several caveats of the method setup should be noticed. To start, the *Flow* model is only verified for three empirically parameterized soil types in a drying situation, i.e. the soil is initially completely saturated and converges to a steady state solution where a constant evaporative surface flux is applied. It can be discussed whether the verification extends to soils with other properties than the current ones tested. It is important to notice that parametrization of the hydraulic conductivity function is based on empirical relations. Although the basis of parametrization is not fundamental, interpolation between the different soil types for which was tested by [Malik et al., 1989] is expected to be valid since the explanatory variables, saturated hydraulic conductivity K_s and wilting point θ_{wp}, can be calculated from continuously defined properties [Alexander and Skaggs, 1987], [Karup et al., 2017] of the soil continuum [Shirazi and Boersma, 1984] which define the relations between the soil types. Therefore, verification of the model with regard to soil types allows for interpolation. Secondly, as a consequence of the model setup only the effect of an evaporative flux on the soil surface is verified. Due to the initial saturation of the soil, gradients in the system increased gradually with decreasing pressure head and approaches the limit of the maximum soil depth incrementally. This is in contrast with a situation where the soil is initially very dry and the top soil receives a certain precipitation flux which would instantly maximally increase the internal pressure head gradients in the system.

Although model verification using a closed form analytical solution is the preferred method, other methods exist such as comparison to another iterative or non-iterative model or using experimental data for model verification [Zhang et al., 2015]. Note that the latter method would be of more significance for verifying a complex test case rather than exact state values since the typical measurement error will probably exceed the desired precision of the numerical scheme.

The method setup of the second case needs some extra explanation to understand its possible limitations. It was chosen to include the depth pressure head soil profiles for all model runs because it is important to have a quantitative measure since the root mean square error (RMSE) statistic alone does not prove a visual fit to the benchmark, e.g. the RMSE does not indicate anything about the error distribution. Nevertheless, the statistic was chosen because its values are expressed in a commonly used and well known unit of length and is therefore convenient to use for simulation comparisons across models. A comparable argument concerning the normalized scalar (absolute) convergence sum averages can be made. Since critical deviations from the benchmark are expected at the top of the domain, where the forcing is applied, distribution of this deviation is not captured and might have the largest concealing effect on the situation with coarsest nodal spacing, although normalized. After all, the number of internal iterations differed between the *Hydrus* and *HydrusPicard* model while implementations should have been identical because the latter model also implements an implicit Euler in time and finite difference in space scheme. However, in the *Hydrus* model extra restrictions were imposed such as the maximum desired absolute change in the value of the water content or pressure head between two successive iterations during a particular time step. Although these restrictions, final stable solution results did not significantly differ between the models.

The method setup of the third case resulted in the possible simplification of the time complexity analysis as the $DMUL$ parameter affected the calculations in much higher degree than $DMUL2$. This result was unintentional. However, it is not expected that this behavior would

have significantly impacted the durations towards solutions between the models as these parameters reflect an opposite or balancing relation in both time stepping schemes. Concerning the simplified duration simulations where the $DMUL$ dependence was examined, normalized standard deviations of *Hydrus* are much larger than the corresponding values for the *Flow* model. This is the case because the absolute values of duration are in the order of one hundredth of a second versus order of a second, respectively. This makes the duration measurements of the *Hydrus* model relatively more prone to other processes that are running in the background on the computer as duration was measured in 'real' time as opposed to 'user' and 'sys' time which indicate the pure CPU time spent. This might have contributed to the larger normalized standard deviations for the run times of the *Hydrus* model but it is not expected that this setup affects the relative differences between the models in terms of run time durations.

The test driven development (TDD) approach resulted in code with a very high testing coverage as each new feature gets assigned at least one test and therefore possible bugs are found relatively early in the development process. Despite these advantages of this development approach, some disadvantages are worth mentioning. The output of a specific method or function is not always exactly clear in advance, especially when output contains a docstring or a table-like structure. In addition, testing every feature of the code might introduce a false sense of testing security, i.e. a feature might pass a test due to a wrongly implemented test case.

The structure of *Flow's* source code is set up such that all tests are included in the source files. This might cause difficulties when files are getting bigger when extending the software during further developments. It is advised to migrate these tests attached in code to separate files for more convenient maintenance. One can always revert to the original state due to the usage of the version control system (VCS) Git [Chacon and Straub, 2014] that was used during the development of the software.

Acknowledgments *I would like to thank my supervisor, dr. ir. Klaas Metselaar for his extensive support; allowing me to take the time needed to enhance my programming skills and for the weekly meetings which kept me on track during the development and writing of my extended MSc thesis. I would also like to thank dr. ir. Jan Wesseling for his valuable feedback on the IT aspect of the study.*

List of Figures

List of Tables

List of codes

References

[Abramowitz and Stegun, 1948] Abramowitz, M. and Stegun, I. A. (1948). *Handbook of mathematical functions with formulas, graphs, and mathematical tables*, volume 55. US Government printing office.

[Alexander and Skaggs, 1987] Alexander, L. and Skaggs, R. W. (1987). Predicting unsaturated hydraulic conductivity from soil texture. *Journal of irrigation and drainage engineering*, 113(2):184–197.

[Atkinson, 2008] Atkinson, K. E. (2008). *An introduction to numerical analysis*. John wiley & sons.

[Backus et al., 1957] Backus, J. W., Beeber, R. J., Best, S., Goldberg, R., Haibt, L. M., Herrick, H. L., Nelson, R. A., Sayre, D., Sheridan, P. B., Stern, H., Ziller, I., Hughes, R. A., and Nutt, R. (1957). The FORTRAN automatic coding system. In *Proceedings of the Western Joint Computer Conference, February 26–28, 1957, Los Angeles, CA, USA*, pages 188–198. The online edition of the Oxford English Dictionary cites this as the second earliest mention of the name FORTRAN, with the extract "The programmer attended a one-day course on FORTRAN and ... then programmed the job in four hours using 47 FORTRAN statements.".

[Baron et al., 2017] Baron, V., Coudière, Y., and Sochala, P. (2017). Adaptive multistep time discretization and linearization based on a posteriori error estimates for the richards equation. *Applied Numerical Mathematics*, 112:104–125.

[Beck, 2003] Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

[Berendsen, 2021] Berendsen, B. (2021). A one-dimensional flow model. `https://github.com/bramappelt/flow`.

[Bishop, 2020] Bishop, S. (2020). *pytz - World Timezone Definitions for Python*. Available at: http://pytz.sourceforge.net/ Accessed: 05-09-2020.

[Brandl, 2010] Brandl, G. (2010). Sphinx documentation. *URL http://sphinx-doc. org/sphinx. pdf*.

[Brandyk and Wesseling, 1985] Brandyk, T. and Wesseling, J. G. (1985). Steady state capillary rise in some soil profiles. *Zeitschrift für Pflanzenernährung und Bodenkunde*, 148(1):54–65.

[Brock and Moore, 2006] Brock, D. C. and Moore, G. E. (2006). *Understanding Moore's law: four decades of innovation*. Chemical Heritage Foundation.

[Brutsaert, 1971] Brutsaert, W. F. (1971). A functional iteration technique for solving the richards equation applied to two-dimensional infiltration problems. *Water Resources Research*, 7(6):1583–1596.

[Celia et al., 1990] Celia, M. A., Bouloutas, E. T., and Zarba, R. L. (1990). A general mass-conservative numerical solution for the unsaturated flow equation. *Water Resources Research*, 26(7):1483–1496.

[Chacon and Straub, 2014] Chacon, S. and Straub, B. (2014). *Pro git*. Apress.

[Coddington and Levinson, 1955] Coddington, E. A. and Levinson, N. (1955). *Theory of ordinary differential equations*. Tata McGraw-Hill Education.

[Darcy, 1856] Darcy, H. P. G. (1856). *Les Fontaines publiques de la ville de Dijon. Exposition et application des principes à suivre et des formules à employer dans les questions de distribution d'eau, etc.* V. Dalamont.

[Eric Gazoni, 2020] Eric Gazoni, C. C. (2020). *openpyxl - A Python library to read/write Excel 2010 xlsx/xlsm files.* Available at: https://openpyxl.readthedocs.io/en/stable/ Accessed: 05-09-2020.

[Farthing and Ogden, 2017] Farthing, M. W. and Ogden, F. L. (2017). Numerical solution of richards' equation: A review of advances and challenges. *Soil Science Society of America Journal*, 81(6):1257–1269.

[Gardner, 1958] Gardner, W. R. (1958). Some steady-state solutions of the unsaturated moisture flow equation with application to evaporation from a water table. *Soil Science*, 85(4):228–232.

[Harris et al., 2020] Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M. H., Brett, M., Haldane, A., Fernández del Río, J., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., and Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, 585:357–362.

[Hunter, 2007] Hunter, J. D. (2007). Matplotlib: A 2d graphics environment. *Computing in Science Engineering*, 9(3):90–95.

[Šimůnek et al., 1992] Šimůnek, J., Vogel, T., and Van Genuchten, M. T. (1992). *SWMS-2D Code for Simulating Water Flow and Solute Transport in Two-dimensional Variably Saturated Media: Version 1.1.* USDA.

[Šimůnek et al., 2013] Šimůnek et al. (2013). The Hydrus-1D Software Package for Simulating the Movement of Water, Heat, and Multiple Solutes in Variably Saturated Media. HYDRUS Software Series 3 Version 4.17, University of California Riverside, Riverside, California, USA.

[ISO, 1995] ISO (1995). *Programming languages — C — Amendment 1: C integrity (ISO/IEC 9899:1990/AMD 1:1995); English version EN 29899:1993/A1:1996 (foreign standard).*

[John D. Hunter, 2020] John D. Hunter, M. D. (2020). *Composable Cycles - Cycler 0.10.0 Documentation.* Available at: https://matplotlib.org/cycler/ Accessed: 05-09-2020.

[Karup et al., 2017] Karup, D., Moldrup, P., Tuller, M., Arthur, E., and de Jonge, L. W. (2017). Prediction of the soil water retention curve for structured soil from saturation to oven-dryness. *European Journal of Soil Science*, 68(1):57–65.

[Kroes et al., 2009] Kroes, J. G., Van Dam, J. C., Groenendijk, P., Hendriks, R. F. a., and Jacobs, C. M. J. (2009). SWAP version 3.2. Theory description and user manual. Technical Report August, Alterra.

[Kwame et al., 2017] Kwame, A. E., Martey, E. M., and Chris, A. G. (2017). Qualitative assessment of compiled, interpreted and hybrid programming languages. *Communications*, 7:8–13.

[Larsbo and Jarvis, 2003] Larsbo, M. and Jarvis, N. (2003). *MACRO 5.0: a model of water flow and solute transport in macroporous soil: technical description.* Department of Soil Sciences, Swedish University of Agricultural Sciences Uppsala.

[List and Radu, 2016] List, F. and Radu, F. A. (2016). A study on iterative methods for solving Richards' equation. *Computational Geosciences*, 20(2):341–353.

[Malik et al., 1989] Malik, R. S., Kumar, S., and Malik, R. K. (1989). Maximal capillary rise flux as a function of height from the water table. *Soil Science*, 148(5):322–326.

[McDonald and Harbaugh, 2003] McDonald, M. G. and Harbaugh, A. W. (2003). The history of MODFLOW. *Ground water*, 41(2):280–283.

[McGuire, 2020] McGuire, P. (2020). *Pyparsing Module.* Available at: https://pyparsing-docs.readthedocs.io/en/latest/pyparsing.html Accessed: 05-09-2020.

[Merelo-Guervós et al., 2016] Merelo-Guervós, J.-J., Blancas-Alvarez, I., Castillo, P. A., Romero, G., Garcıa-Sánchez, P., Rivas, V. M., Garcıa-Valdez, M., Hernández-Aguila, A., and Román, M. (2016). Ranking the performance of compiled and interpreted languages in genetic algorithms. In *International Conference on Evolutionary Computation Theory and Applications*, volume 2, pages 164–170. SCITEPRESS.

[Miller et al., 1998] Miller, C. T., Christakos, G., Imhoff, P. T., McBride, J. F., Pedit, J. A., and Trangenstein, J. A. (1998). Multiphase flow and transport modeling in heterogeneous porous media: challenges and approaches. *Advances in Water Resources*, 21(2):77–120.

[Miller et al., 2013] Miller, C. T., Dawson, C. N., Farthing, M. W., Hou, T. Y., Huang, J., Kees, C. E., Kelley, C., and Langtangen, H. P. (2013). Numerical simulation of water resources problems: Models, methods, and trends. *Advances in Water Resources*, 51:405–437.

[Mls, 1982] Mls, J. (1982). Formulation and solution of fundamental problems of vertical infiltration. *Vodohosp. Cas*, 30:304–311.

[Mostaghimi et al., 2015] Mostaghimi, P., Percival, J. R., Pavlidis, D., Ferrier, R. J., Gomes, J. L., Gorman, G. J., Jackson, M. D., Neethling, S. J., and Pain, C. C. (2015). Anisotropic mesh adaptivity and control volume finite element methods for numerical simulation of multiphase flow in porous media. *Mathematical Geosciences*, 47(4):417–440.

[Niemeyer, 2020] Niemeyer, G. (2020). *Dateutil - Powerful extension to datetime.* Available at: https://dateutil.readthedocs.io/en/stable/ Accessed: 05-09-2020.

[Nucleic, 2020] Nucleic (2020). *Kiwisolvers's Documentation.* Available at: https://kiwisolver.readthedocs.io/en/latest/ Accessed: 05-09-2020.

[Ogden et al., 2017] Ogden, F. L., Allen, M. B., Lai, W., Zhu, J., Seo, M., Douglas, C. C., and Talbot, C. A. (2017). The soil moisture velocity equation. *Journal of Advances in Modeling Earth Systems*, 9(2):1473–1487.

[Peterson, 2020] Peterson, B. (2020). *Six: Python2 and 3 Compatibility Library.* Available at: https://six.readthedocs.io/ Accessed: 05-09-2020.

[R. H. Brooks & A. T. Corey, 1964] R. H. Brooks & A. T. Corey (1964). Hydraulic Properties of Porous Media and Their Relation to Drainage Design. *Transactions of the ASAE*, 7(1):0026–0028.

[Richards, 1931] Richards, L. A. (1931). Capillary conduction of liquids through porous mediums. *Physics*, 1(5):318–333.

[Shirazi and Boersma, 1984] Shirazi, M. A. and Boersma, L. (1984). A unifying quantitative analysis of soil texture. *Soil Science Society of America Journal*, 48(1):142–147.

[Šimůnek et al., 2008] Šimůnek, J., van Genuchten, M. T., and Šejna, M. (2008). Development and Applications of the HYDRUS and STANMOD Software Packages and Related Codes. *Vadose Zone Journal*, 7(2):587–600.

[Srinath, 2017] Srinath, K. (2017). Python–the fastest growing programming language. *International Research Journal of Engineering and Technology*, 4(12):354–357.

[van Genuchten, 1980] van Genuchten, M. T. (1980). A Closed-form Equation for Predicting the Hydraulic Conductivity of Unsaturated Soils. *Soil Science Society of America Journal*, 44(5):892–898.

[Van Rossum and Drake, 2009] Van Rossum, G. and Drake, F. L. (2009). *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA.

[Vanclooster et al., 1996] Vanclooster, M., Viaene, P., Diels, J., and Christiaens, K. (1996). WAVE: A mathematical model for simulating water and agrochemicals in the soil and vadose environment, Release 2.1. *Wave Reference Manual*, (January):145pp.

[Vanderborght et al., 2005] Vanderborght, J., Kasteel, R., Herbst, M., Javaux, M., Thiery, D., Vanclooster, M., Mouvet, C., and Vereecken, H. (2005). A Set of Analytical Benchmarks to Test Numerical Models of Flow and Transport in Soils. *Vadose Zone Journal*, 4(1):206–221.

[Virtanen et al., 2020] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Millman, K. J., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C. J., Polat, İ., Feng, Y., Moore, E. W., VanderPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and SciPy 1.0 Contributors (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

[Vogel et al., 1996] Vogel, T., Huang, K., Zhang, R., and Van Genuchten, M. T. (1996). The hydrus code for simulating one-dimensional water flow, solute transport, and heat movement in variably-saturated media. *US Salinity Lab, Riverside, CA*.

[Wes McKinney, 2010] Wes McKinney (2010). Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61.

[Wösten et al., 2001] Wösten, J. H. M., Veerman, G., DeGroot, W., and Stolte, J. (2001). Waterretentie- en doorlatendheidskarakteristieken van boven- en ondergronden in Nederland: de Staringreeks. *Alterra Rapport*, 153:86.

[Ypma, 1995] Ypma, T. J. (1995). Historical Development of the Newton-Raphson Method Author (s): Tjalling J . Ypma Reviewed work (s): Published by : Society for Industrial and Applied Mathematics Stable URL : http://www.jstor.org/stable/2132904 . *Society for Industrial and Applied Mathematics*, 37(4):531–551.

[Zha et al., 2019] Zha, Y., Yang, J., Zeng, J., Tso, C.-H. M., Zeng, W., and Shi, L. (2019). Review of numerical solution of richardson–richards equation for variably saturated flow in soils. *Wiley Interdisciplinary Reviews: Water*, 6(5):e1364.

[Zhang et al., 2015] Zhang, Z., Wang, W., Chen, L., Zhao, Y., An, K., Zhang, L., and Liu, H. (2015). Finite analytic method for solving the unsaturated flow equation. *Vadose Zone Journal*, 14(1):1–10.

A Appendices

A.1 Appendix - Argument map

Table 14: Argument mapper that maps corresponding parameters of *Flow* and *Hydrus* that are used and varied in the model simulation study. The exact definitions of the arguments can be found in *Application Programming Interface (API) Documentation* and [Šimůnek et al., 2013] for the models respectively. Note that some input parameters are calculated implicitly (*) or have dedicated functions (**) instead of a single argument name.

In-text	*Flow*	*Hydrus*
dt	dt	dt
$dtMin$	dt_min	$dtMin$
$dtMax$	dt_max	$dtMax$
$ItMin$	$itermin$	$ItMin$
$ItMax$	$itermax$	$ItMax$
$DMUL$	$dtitlow$	$dMul$
$DMUL2$	$dtithigh$	$dMul2$
$MaxIt$	$maxiter$	$MaxIt$
$tMax$	$endtime$	$tMax$
$threshold$	$threshold$	$TolTh$
L	$*$	$xFix2$
N	$numnodes$	$NumNP$
$Power$	$power$	$-$
s_{init}	$initial_states$	$hNew$
$rTop$	**	$rTop$
BC_{bot}	**	$KodBot$
BC_{top}	**	$KodTop$
$soiltype$	$*$	$*$

A.2 Appendix - Case 1

A.2.1 Conversion deviations

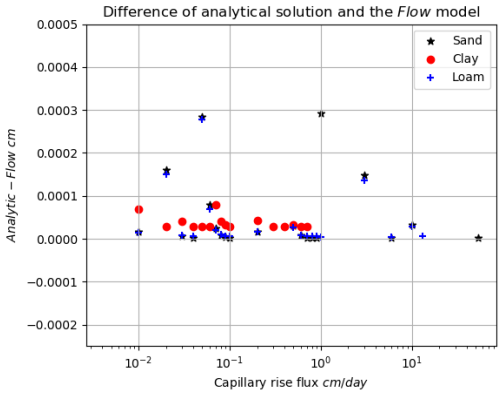


Figure 16: Case 1 - Differences between analytical solution of [Malik et al., 1989] and the *Flow* model for three different soil types.

A.3 Appendix - Case 2

A.3.1 Cases' source code

```
import os
import functools

import numpy as np

from waterflow.flow1d.flowFE1d import Flow1DFE
from waterflow.utility import conductivityfunctions as condf
from waterflow.utility import fluxfunctions as fluxf
from waterflow.utility.spacing import biasedspacing
from waterflow.utility.helper import initializer, converged

import sys
sys.path.append('C:\\Users\\bramb\\OneDrive\\thesis') # noqa

from hydrus_parser.selectorINsetup import selector_in # noqa
from hydrus_parser.profileDATsetup import profile_dat # noqa
from hydrus_parser.hydrusoutput import get_all_hydrus_frames # noqa


################################################################################
################################ FLOW TEST MODE1 ###############################
################################################################################

class Flow(Flow1DFE):
    '''Prepare and execute a Flow simulation'''

    def __init__(self, options, **kwargs):
        '''named arguments passed via kwargs have precedence'''

        super().__init__(id_=None)
        self.options = dict(options)
        self.options.update(**kwargs)
        self.run()
        self.idx_nr_states = np.nancumsum(
            self.dft_solved_times.iter, dtype=int)

    def run(self, options=None):
        '''run the model calculation'''

        opts = options or self.options

        # discretization
        xsp = biasedspacing(numnodes=opts.get('N'),
                            power=opts.get('power'),
                            maxdist=opts.get('maxdist'),
                            rb=opts.get('L')) * -1

        # initial states
        initial_states = opts.get('H_init')
        if not isinstance(initial_states, (int, float)):
            initial_states = initial_states[::-1]

        # soil water retention parameterization
        conductivity_func = initializer(condf.VG_conductivity,
                                        ksat=opts.get('Ks'),
                                        a=opts.get('Alfa'),
                                        n=opts.get('n'))

```

```
        theta_h = initializer(condf.VG_pressureh,
                              a=opts.get('Alfa'),
                              n=opts.get('n'),
                              theta_r=opts.get('thr'),
                              theta_s=opts.get('ths'))

        storage_change = initializer(fluxf.storage_change, fun=theta_h)

        # Flow's general model setup
        # This part is altered to suit the setup of any test case
        self.set_field1d(nodes=xsp[::-1])
        self.set_gaussian_quadrature(opts.get('gauss_degree'))
        self.set_initial_states(initial_states)

        self.set_systemfluxfunction(
            fluxf.richards_equation, kfun=conductivity_func)

        self.add_dirichlet_BC(opts.get('bounds')[-1], 'west')
        self.add_dirichlet_BC(opts.get('bounds')[0], 'east')
        self.add_spatialflux(storage_change)

        # solve procedure
        self.solve(dt=opts.get('dt'),
                   dt_min=opts.get('dtMin'),
                   dt_max=opts.get('dtMax'),
                   end_time=opts.get('tMax'),
                   maxiter=opts.get('MaxIt'),
                   itermin=opts.get('ItMin'),
                   itermax=opts.get('ItMax'),
                   threshold=opts.get('threshold'),
                   verbosity=True,
                   storeNR=True)

        self.dataframeify(invert=True)
        self.transient_dataframeify()


################################################################################
############################## HYDRUS TEST MODEL ###############################
################################################################################

class Hydrus:
    '''Prepare and execute a Hydrus simulation'''

    selector = 'SELECTOR.IN'
    profile = 'PROFILE.DAT'

    def __init__(self, sopts, popts=None, directory=''):
        self.sopts = sopts
        self.popts = popts or {}
        self.directory = directory

    def prepare_new_model(self, name):
        self.runpath = os.path.join(self.directory, name)
        self.selector_in = os.path.join(self.runpath, self.selector)
        self.profile_dat = os.path.join(self.runpath, self.profile)

        if not os.path.isdir(self.runpath):
            os.mkdir(self.runpath)

    def build_selector_in(self):
        '''populate "SELECTOR.IN" with custom options'''
```

```
        self._internal_sopts = selector_in(self.selector_in, self.sopts)

    def build_profile_dat(self):
        '''populate "PROFILE.DAT" with custom options'''

        # add spacing
        N = self.sopts.get('N')
        power = self.sopts.get('power')
        maxdist = self.sopts.get('maxdist')
        L = self.sopts.get('L')

        xsp = biasedspacing(
            numnodes=N, power=power, maxdist=maxdist, rb=L) * -1

        # add initial states
        states = self.sopts.get('H_init')
        if isinstance(states, (int, float)):
            states = np.repeat(states, N)

        # dirichlet bc if set
        bounds = self.sopts.get('bounds')
        if bounds:
            states[[0, -1]] = bounds

        self.popts.update({'NumNP': N,
                           'iFix2': -L,
                           'x': xsp,
                           'h': states})
        self._internal_popts = profile_dat(self.profile_dat, self.popts)

    def execute(self):
        '''start simulation for pre-set values'''
        os.system('H1D_CALC.exe {}'.format(self.runpath))

    def run(self):
        '''prepare, simulate and fetch results'''

        self.build_selector_in()
        self.build_profile_dat()
        self.execute()
        self.get_frames()

    def get_frames(self):
        '''parse frames from text files build by Hydrus'''
        self.frames = get_all_hydrus_frames(self.runpath)

    def new_model(self, name, **options):
        '''new simulation using predefined defaults'''

        sopts = dict(self.sopts)
        popts = dict(self.popts)
        sopts.update(options)

        new_model = Hydrus(sopts, popts, directory=self.directory)
        new_model.prepare_new_model(name)
        new_model.run()

        return new_model


################################################################################
```

```
######################## HYDRUS PICARD ITERATION TEST MODEL ######################
##################################################################################


def vg_t_h(h, a, n, theta_r, theta_s):
    '''Van Genuchten 1980'''
    m = 1 - 1/n
    return (theta_s - theta_r) / (1 + (a * abs(h)**n)**m) + theta_r


def vg_k(h, ksat, a, n):
    '''Van Genuchten 1980'''
    m = 1 - 1/n
    up = (1 - (a * abs(h)) ** (n - 1) * (1 + (a * abs(h))**n)**-m)**2
    down = (1 + (a * abs(h))**n) ** (m/2)
    return ksat * up / down


def C(h, theta, dh):
    '''Soil Moisure Capacity'''
    return (theta(h + dh * 0.5) - theta(h - dh * 0.5)) / dh


class HydrusPicard:
    '''Simulation of hydrus' implemented picard scheme iteration'''

    def __init__(self, options, **kwargs):
        '''named arguments passed via kwargs have precedence'''

        self.options = dict(options)
        self.options.update(**kwargs)
        self.run()

    def set_soilmodel(self, ksat, a, n, theta_r, theta_s, dh):
        '''parameterize soil model functions'''

        self.kfun = functools.partial(vg_k, ksat=ksat, a=a, n=n)

        self.tfun = functools.partial(
            vg_t_h, a=a, n=n, theta_r=theta_r, theta_s=theta_s)

        self.cfun = functools.partial(C, theta=self.tfun, dh=dh)

    def set_initial_states(self, states, bcs):
        '''set initial states'''

        if isinstance(states, (int, float)):
            self.initial_states = np.repeat(states, self.N)
        else:
            self.initial_states = states

        self.initial_states[[0, -1]] = bcs

        self.bcs = np.zeros(self.N)
        self.bcs[[0, -1]] = bcs

    def discretize(self, z):
        '''nodal discretization'''

        N = len(z)
        L = z[-1]

```

```
        dz = np.repeat(0, N).astype(np.float64)
        for i in range(N):
            if i == 0:
                dz[i] = z[i+1] / 2
            elif i == N - 1:
                dz[i] = (z[i] - z[i-1]) / 2
            else:
                dz[i] = (z[i+1] - z[i-1]) / 2

        self.N = N
        self.L = L
        self.z = z
        self.dz = dz

    def matrixFD(self, H, dt):
        '''implicit Euler in time, finite differences in space coefficient
           matrix'''

        A = np.zeros((self.N, self.N)).astype(np.float64)
        A2 = np.zeros(self.N)
        A3 = np.zeros((self.N, self.N))
        for i in range(1, self.N - 1):
            kleft = 0.5 * (self.kfun(H[i-1]) + self.kfun(H[i]))
            kright = 0.5 * (self.kfun(H[i]) + self.kfun(H[i+1]))
            kmid = -kleft - kright

            dz2dt = -self.dz[i] ** 2 / dt

            A[i, i-1:i+2] = np.array([kleft, kmid + self.cfun(H[i]) * dz2dt,
    kright])
            A2[i] = -self.dz[i] * (kright - kleft)
            A3[i, i] = dz2dt

        # dirichlet bc
        A[0][0] = 1
        A[-1][-1] = 1

        return A, A2, A3

    def solve(self, dt, MatIt, threshold, tMax=None, based='t'):
        '''solve Richards equation'''

        self.dt_states = [self.initial_states]
        self.picard_states = []
        self.picard_iters = [(np.nan, np.nan)]

        time = 0
        while time < (tMax or dt):
            self.picard_states.append(self.dt_states[-1])

            H_new = self.picard_states[-1]
            H_old = H_new

            niter = 0
            convergence = False
            while niter < MatIt and not convergence:

                A, b, c = self.matrixFD(H_new, dt)

                if based == 'h':
                    temporal = np.matmul(c * self.cfun(H_new), H_old)

```

```
                if based == 't':
                    temporal = np.matmul(c * self.cfun(H_new), H_new) - \
                            np.matmul(c, self.tfun(H_new) - self.tfun(H_old))

                H_previous = H_new
                H_new = np.linalg.solve(A, self.bcs + b + temporal)

                convergence = converged(H_previous, H_new, threshold)

                niter += 1
                self.picard_states.append(H_new)

            time += dt
            self.picard_iters.append((time, niter))
            self.dt_states.append(H_new)

    def run(self, options=None):
        '''run the model calculation'''

        opts = options or self.options

        # discretization
        z = biasedspacing(numnodes=opts.get('N'),
                          power=opts.get('power'),
                          maxdist=opts.get('maxdist'),
                          rb=opts.get('L')) * -1

        self.discretize(z=z)

        # soil water retention parameterization
        self.set_soilmodel(ksat=opts.get('Ks'),
                           a=opts.get('Alfa'),
                           n=opts.get('n'),
                           theta_r=opts.get('thr'),
                           theta_s=opts.get('ths'),
                           dh=opts.get('dh'))

        # initial states
        self.set_initial_states(states=opts.get('H_init'),
                                bcs=opts.get('bounds'))

        # solve procedure
        self.solve(dt=opts.get('dt'),
                   MatIt=opts.get('MaxIt'),
                   threshold=opts.get('threshold'),
                   tMax=opts.get('tMax'),
                   based=opts.get('based'))
```

Code 1: Case 2 - Source code.

A.4 Appendix - Case 3

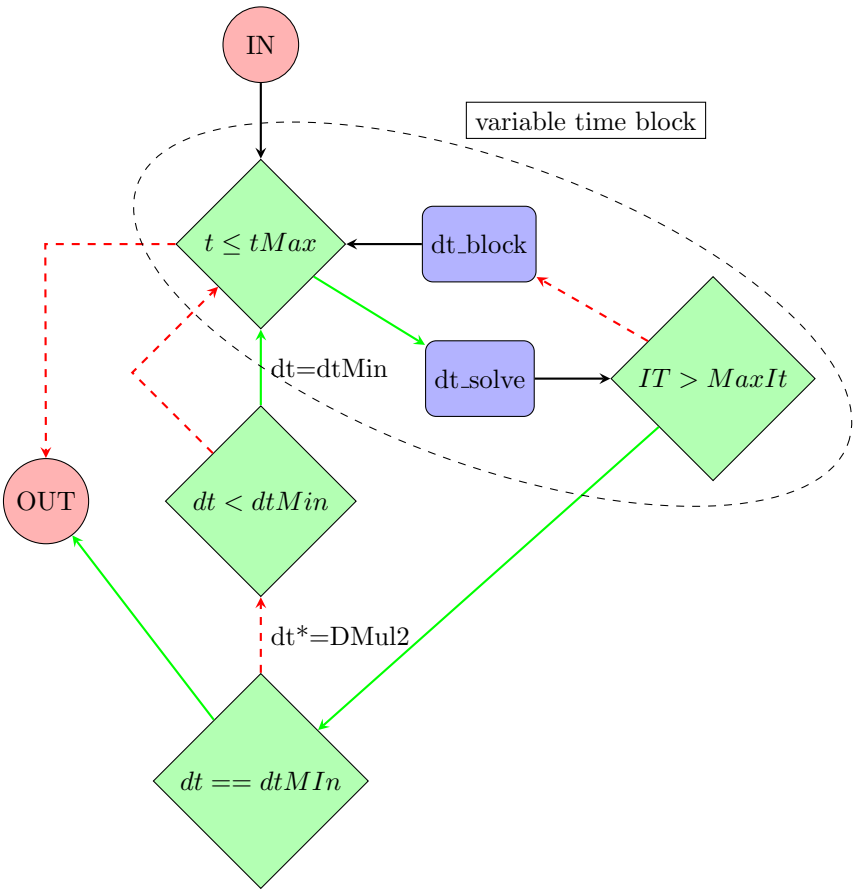
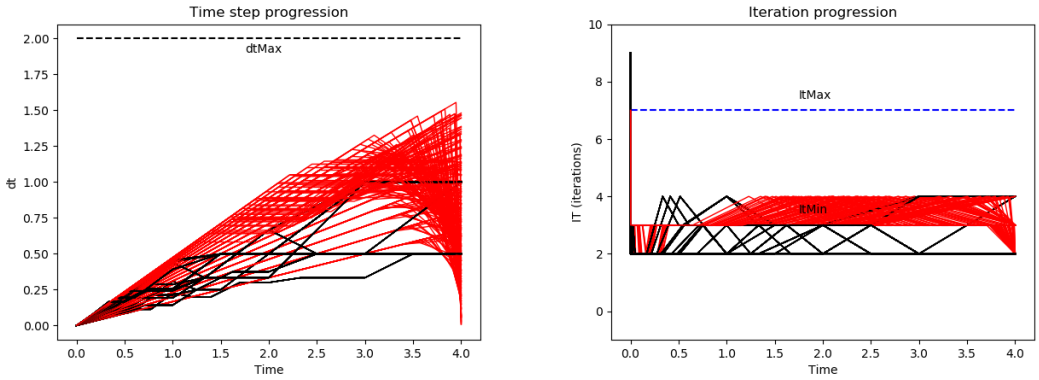


Figure 17: Case 3 - Complete schematic solve procedure of *Flow*. Arrows indicate the direction of flow in the diagram when conditions evaluate positive (green) or negative (red).



(a) Time step (dt) progression and the maximum allowed time step size ($dtMax$).

(b) Number of iterations (IT) including lower ($ItMin$) and upper ($ItMax$) bounds.

Figure 18: Case 3 - Time complexity constraints for all model runs of *Flow* (red) and *Hydrus* (black). Note that dt and $time$ are in terms of days.

A.5 Appendix - Hydrus schemes

A.5.1 Forms of Richards' equations

h-based

$$C(h)\frac{\partial h}{\partial t} - \nabla \cdot K(h)\nabla h - \frac{\partial K}{\partial z} = 0 \tag{47}$$

θ-based

$$\frac{\partial \theta}{\partial t} - \nabla \cdot D(\theta)\nabla \theta - \frac{\partial K}{\partial z} = 0 \tag{48}$$

θ/h mixed

$$\frac{\partial \theta}{\partial t} - \nabla \cdot K(h)\nabla h - \frac{\partial K}{\partial z} = 0 \tag{49}$$

A.5.2 Euler backward in time

h-based

$$C^{n+1}\frac{H^{n+1} - H^n}{\Delta t} - \nabla \cdot K^{n+1}\nabla H^{n+1} - \frac{\partial K^{n+1}}{\partial z} = 0 \tag{50}$$

θ/h mixed

$$\frac{\theta^{n+1} - \theta^n}{\Delta t} - \nabla \cdot K^{n+1}\nabla H^{n+1} - \frac{\partial K^{n+1}}{\partial z} = 0 \tag{51}$$

A.5.3 Picard iteration scheme

h-based

$$C^{n+1,m}\frac{H^{n+1,m+1}-H^{n}}{\Delta t}-\nabla\cdot K^{n+1,m}\nabla H^{n+1,m+1}-\frac{\partial K^{n+1,m}}{\partial z}=0 \tag{52}$$

Rewrite

$$\begin{aligned} C^{n+1,m}\frac{H^{n+1,m+1}-H^{n+1,m}}{\Delta t}&-\nabla\cdot K^{n+1,m}\nabla(H^{n+1,m+1}-H^{n+1,m}) \\ &= \\ -C^{n+1,m}\frac{H^{n+1,m}-H^{n}}{\Delta t}&+\nabla\cdot K^{n+1,m}\nabla H^{n+1,m}+\frac{\partial K^{n+1,m}}{\partial z} \\ &= \\ &R^{n+1,m}_{Picard} \end{aligned} \tag{53}$$

θ/h mixed

$$\frac{\theta^{n+1,m+1}-\theta^{n}}{\Delta t}-\nabla\cdot K^{n+1,m}\nabla H^{n+1,m+1}-\frac{\partial K^{n+1,m}}{\partial z}=0 \tag{54}$$

where $\theta^{n+1,m+1}$ is approximated by a Taylor series expansion.

$$\theta^{n+1,m+1}=\theta^{n+1,m}+\frac{\partial\theta}{\partial h}^{n+1,m}(H^{n+1,m+1}-H^{n+1,m}) \tag{55}$$

which reads as follows after subsitution.

$$\begin{aligned} C^{n+1,m}\frac{H^{n+1,m+1}-H^{n+1,m}}{\Delta t}&+\frac{\theta^{n+1,m}-\theta^{n}}{\Delta t} \\ &-\nabla\cdot K^{n+1,m}\nabla H^{n+1,m+1}-\frac{\partial K^{n+1,m}}{\partial z}=0 \end{aligned} \tag{56}$$

Rewrite

$$\begin{aligned} C^{n+1,m}\frac{H^{n+1,m+1}-H^{n+1,m}}{\Delta t}&-\nabla\cdot K^{n+1,m}\nabla(H^{n+1,m+1}-H^{n+1,m}) \\ &= \\ -C^{n+1,m}\frac{H^{n+1,m}-H^{n}}{\Delta t}&+\nabla\cdot K^{n+1,m}\nabla H^{n+1,m}+\frac{K^{n+1,m}}{\partial z}-\frac{\theta^{n+1,m}-\theta^{n}}{\Delta t} \\ &= \\ &R^{n+1,m}_{ModifiedPicard} \end{aligned} \tag{57}$$

A.5.4 Finite Difference Approximation

h-based Apply finite difference (FD) approximation to eq. (53).

$$\begin{aligned} & C^{n+1,m}\frac{\psi_i^m}{\Delta t} \\ & -\frac{1}{(\Delta z)^2}[K_{i+1/2}^{n+1,m}(\psi_{i+1}^m-\psi_i^m)-K_{i-1/2}^{n+1,m}(\psi_i^m-\psi_{i-1}^m)] = \\ & -C_i^{n+1,m}\frac{H_i^{n+1,m}-H_i^n}{\Delta t} \\ & +\frac{1}{(\Delta z)^2}[K_{i+1/2}^{n+1,m}(H_{i+1}^{n+1,m}-H_i^{n+1,m})-K_{i-1/2}^{n+1,m}(H_i^{n+1,m}-H_{i-1}^{n+1,m})] \\ & +\frac{K_{i+1/2}^{n+1,m}-K_{i-1/2}^{n+1,m}}{\Delta z} = \\ & (R_i^{n+1,m})_{PicardFD} \end{aligned} \tag{58}$$

Rewrite the center and right terms of eq. (58).

$$\begin{aligned} & \frac{1}{(\Delta z)^2}[H_{i-1}K_{i-1/2}+H_i(-K_{i-1/2}-K_{i+1/2})+H_{i+1}K_{i+1/2}]^{n+1,m} = \\ & -\frac{1}{\Delta z}[K_{i+1/2}-K_{i-1/2}]^{n+1,m}+\frac{C_i^{n+1,m}}{\Delta t}[H_i]^{n+1,m}-\frac{C_i^{n+1,m}}{\Delta t}[H_i]^n \end{aligned} \tag{59}$$

Move factors and add transient term to coëfficiënt matrix.

$$\begin{aligned} & [H_{i-1}K_{i-1/2}+H_i(-K_{i-1/2}-K_{i+1/2}-C_i^{n+1,m}\frac{\Delta z^2}{\Delta t})+H_{i+1}K_{i+1/2}]^{n+1,m} = \\ & -\Delta z[K_{i+1/2}-K_{i-1/2}]^{n+1,m}-C_i^{n+1,m}\frac{\Delta z^2}{\Delta t}[H_i]^n \end{aligned} \tag{60}$$

The compact form reads as follows:

$$A\cdot[H_i]^{n+1,m} = -\Delta z[K]^{n+1,m}-\frac{\Delta z^2}{\Delta t}C_i^{n+1,m}[H_i]^n \tag{61}$$

θ/h **mixed** Apply finite difference (FD) approximation to eq. (57).

$$\begin{aligned} C^{n+1,m}\frac{\psi_i^m}{\Delta t} \\ -\frac{1}{(\Delta z)^2}[K_{i+1/2}^{n+1,m}(\psi_{i+1}^m-\psi_i^m)-K_{i-1/2}^{n+1,m}(\psi_i^m-\psi_{i-1}^m)] = \\ -C_i^{n+1,m}\frac{H_i^{n+1,m}-H_i^n}{\Delta t} \\ +\frac{1}{(\Delta z)^2}[K_{i+1/2}^{n+1,m}(H_{i+1}^{n+1,m}-H_i^{n+1,m})-K_{i-1/2}^{n+1,m}(H_i^{n+1,m}-H_{i-1}^{n+1,m})] \\ +\frac{K_{i+1/2}^{n+1,m}-K_{i-1/2}^{n+1,m}}{\Delta z}-\frac{\theta_i^{n+1,m}-\theta_i^n}{\Delta t} = \\ (R_i^{n+1,m})_{ModifiedPicardFD} \end{aligned} \tag{62}$$

Rewrite the center and right terms of eq. (62).

$$\begin{aligned} \frac{1}{(\Delta z)^2}[H_{i-1}K_{i-1/2}+H_i(-K_{i-1/2}-K_{i+1/2})+H_{i+1}K_{i+1/2}]^{n+1,m} = \\ -\frac{1}{\Delta z}[K_{i+1/2}-K_{i-1/2}]^{n+1,m} \\ +\frac{C_i^{n+1,m}}{\Delta t}[H_i]^{n+1,m}-\frac{C_i^{n+1,m}}{\Delta t}[H_i]^n+\frac{\theta_i^{n+1,m}}{\Delta t}-\frac{\theta_i^n}{\Delta t} \end{aligned} \tag{63}$$

Move factors and add transient term to coëfficiënt matrix.

$$\begin{aligned} [H_{i-1}K_{i-1/2}+H_i(-K_{i-1/2}-K_{i+1/2}-C_i^{n+1,m}\frac{\Delta z^2}{\Delta t})+H_{i+1}K_{i+1/2}]^{n+1,m} = \\ -\Delta z[K_{i+1/2}-K_{i-1/2}]^{n+1,m}+\frac{\Delta z^2}{\Delta t}(\theta_i^{n+1,m}-\theta_i^n-C_i^{n+1,m}[H_i]^n) \end{aligned} \tag{64}$$

The compact form reads as follows:

$$A\cdot[H_i]^{n+1,m} = -\Delta z[K]^{n+1,m}+\frac{\Delta z^2}{\Delta t}(\theta_i^{n+1,m}-\theta_i^n-C_i^{n+1,m}[H_i]^n) \tag{65}$$

A.5.5 Finite Element Approximation

***h*-based** Apply finite element (FE) approximation to eq. (53)

$$\begin{aligned}
&-\bar{C}_{i-1}\frac{H_{i-1}^{n+1,m}-H_{i-1}^{n}}{\Delta t}-\bar{C}_{i}\frac{H_{i}^{n+1,m}-H_{i}^{n}}{\Delta t}-\bar{C}_{i+1}\frac{H_{i+1}^{n+1,m}-H_{i+1}^{n}}{\Delta t}\\
&+\frac{1}{(\Delta z)^2}[K_{i+1/2}^{n+1,m}(H_{i+1}^{n+1,m}-H_{i}^{n+1,m})-K_{i-1/2}^{n+1,m}(H_{i}^{n+1,m}-H_{i-1}^{n+1,m})]\\
&+\frac{K_{i+1/2}^{n+1,m}-K_{i-1/2}^{n+1,m}}{\Delta z}=\\
&(R_{i}^{n+1,m})_{PicardFE}
\end{aligned}$$

where

$$\begin{aligned}
\bar{C}_{n-1}&=\frac{1}{12}(C_{i-1}^{n+1,m}+C_{i}^{n+1,m})\\
\bar{C}_{n}&=\frac{1}{12}(C_{i-1}^{n+1,m}+6C_{i}^{n+1,m})+C_{i+1}^{n+1,m}\\
\bar{C}_{n-1}&=\frac{1}{12}(C_{i}^{n+1,m}+C_{i+1}^{n+1,m})
\end{aligned}\tag{66}$$

Rewrite

$$\begin{aligned}
&\frac{1}{(\Delta z)^2}[K_{i-1/2}H_{i-1}+(-K_{i-1/2}-K_{i+1/2})H_{i}+K_{i+1/2}H_{i+1}]^{n+1,m}=\\
&-\frac{1}{\Delta z}[K_{i+1/2}-K_{i-1/2}]^{n+1,m}\\
&\frac{1}{\Delta t}[\bar{C}_{i-1}H_{i-1}+\bar{C}_{i}H_{i}+\bar{C}_{i+1}H_{i+1}]^{n+1,m}\\
&-\frac{1}{\Delta t}[\bar{C}_{i-1}H_{i-1}+\bar{C}_{i}H_{i}+\bar{C}_{i+1}H_{i+1}]^{n}
\end{aligned}\tag{67}$$

Move factors and add transient term to coëfficiënt matrix.

$$\begin{aligned}
&[(K_{i-1/2}-\frac{\Delta z^2}{\Delta t}\bar{C}_{i-1})H_{i-1}+(-K_{i-1/2}-K_{i+1/2}-\frac{\Delta z^2}{\Delta t}\bar{C}_{i})H_{i}\\
&+(K_{i+1/2}-\frac{\Delta z^2}{\Delta t}\bar{C}_{i+1})H_{i+1}]^{n+1,m}=\\
&-\Delta z[K_{i+1/2}-K_{i-1/2}]^{n+1,m}\\
&-\frac{\Delta z^2}{\Delta t}[\bar{C}_{i-1}H_{i-1}+\bar{C}_{i}H_{i}+\bar{C}_{i+1}H_{i+1}]^{n}
\end{aligned}\tag{68}$$

The compact form reads as follows:

$$A\cdot[H_i]^{n+1,m}=-\Delta z[K]^{n+1,m}-\frac{\Delta z^2}{\Delta t}B\cdot[H_i]^{n}\tag{69}$$

A.6 Appendix - *Flow* Documentation

The *Flow* documentation and source code is available online at GitHub [Berendsen, 2021].

A.6.1 Glossary

See the Chapter Three, Symbol glossary, in *Application Programming Interface (API) Documentation.*

A.6.2 Application Programming Interface (API) Documentation

The documentation is hosted online.

A.6.3 Dependencies

Table 15: *Flow's* dependencies.

Dependency	Version	Reference
cycler	0.10.0	[John D. Hunter, 2020]
kiwisolver	1.1.0	[Nucleic, 2020]
matplotlib	3.1.2	[Hunter, 2007]
numpy	1.18.0	[Harris et al., 2020]
pandas	0.25.3	[Wes McKinney, 2010]
pyparsing	2.4.5	[McGuire, 2020]
python-dateutil	2.8.1	[Niemeyer, 2020]
pytz	2019.3	[Bishop, 2020]
scipy	1.4.1	[Virtanen et al., 2020]
six	1.13.0	[Peterson, 2020]
openpyxl	3.0.5	[Eric Gazoni, 2020]

A.6.4 Model's source code

```
from inspect import signature
from functools import partial
from copy import deepcopy
import time as Time
import os

import numpy as np
import pandas as pd
from scipy.special import legendre

from waterflow import OUTPUT_DIR
from waterflow.utility.helper import converged


class Flow1DFE:
    def __init__(self, id_, savepath=OUTPUT_DIR):
        self.id_ = id_
        self.savepath = savepath
        self.systemfluxfunc = None
        self.nodes = None
        self.states = None
        self.nr_states = []
        self.seg_lenghts = None
        self.lengths = None
        self.coefmatr = None
        self.BCs = {}
        self.pointflux = {}
        self.spatflux = {}
        self.Spointflux = {}
        self.Sspatflux = {}
        self.internal_forcing = {}
        self.forcing = None
        self.conductivities = []
        self.moisture = []
        self.fluxes = []
        self.isinitial = True
        self.isconverged = False
        self.solve_data = None
        self.runtime = None
        self.summarystring = ""
        # dataframes
        self.df_states = None
        self.df_balance = None
        self.df_balance_summary = None
        self.dft_solved_times = None
        self.dft_print_times = None
        self.dft_states = None
        self.dft_nodes = None
        self.dft_balance = None
        self.dft_balance_summary = None
        # private attributes
        self._west = None
        self._east = None
        self._delta = 1e-5
        # Gauss specific
        self.gauss_degree = 1
        self._xgauss = None
        self._wgauss = None
        self.gaussquad = None
        self.xintegration = None
```

```
61
62      def __repr__(self):
63          return "Flow1DFE(" + str(self.id_) + ")"
64
65      def summary(self, show=True, save=False, path=None):
66          # build key : value pairs where data is available
67          id_ = f"{self.id_}"
68          if self.nodes is not None:
69              len_ = str(self.nodes[-1] - self.nodes[0])
70              num_nodes = str(len(self.nodes))
71          else:
72              len_ = None
73              num_nodes = None
74          degree = self.gauss_degree
75          bcs = [[k, self.BCs[k][0], self.BCs[k][1]] for k in self.BCs.keys()]
76          bcs = ["{} value: {} and of type {}".format(*bc) for bc in bcs]
77          bcs = ", ".join(i for i in bcs)
78          pkeys = list(self.pointflux.keys()) + list(self.Spointflux.keys())
79          skeys = list(self.spatflux.keys()) + list(self.Sspatflux.keys())
80          pointflux = ", ".join(i for i in pkeys)
81          spatflux = ", ".join(i for i in skeys)
82          runtime = self.runtime
83
84          if hasattr(self, 'kfun'):
85              kfun = self.kfun.__name__
86          else:
87              kfun = None
88
89          if hasattr(self, 'tfun'):
90              tfun = self.tfun.__name__
91          else:
92              tfun = None
93
94          k = ['Id', 'System length', 'Number of nodes', 'Gauss degree',
95               'kfun', 'tfun', 'BCs', 'Pointflux', 'Spatflux', 'Runtime (s)']
96          v = (id_, len_, num_nodes, degree, kfun, tfun, bcs, pointflux,
97               spatflux, runtime)
98
99          # build summary string
100         sumstring = ""
101         for i, j in zip(k, v):
102             if j:
103                 sumstring += f"{i}: {j}\n"
104
105         try:
106             self.calcbalance()
107             sumstring += '\n' + self.df_balance_summary.to_string()
108         except Exception:
109             pass
110
111         # print to console
112         if show:
113             for s in sumstring.split('\n'):
114                 print(s)
115
116         # save to disk
117         if save:
118             if not os.path.isdir(path):
119                 os.mkdir(path)
120
121             fname = f"{self.id_}.txt"
122             with open(os.path.join(path, fname), "w") as fw:
```

```
                fw.write(sumstring)

        self.summarystring = sumstring

    def set_gaussian_quadrature(self, degree=1):
        # calculate roots of Legendre polynomial for degree n
        legn = legendre(degree)
        roots = np.sort(legn.r)

        # calculate corresponding weights
        weights = 2 / ((1 - roots**2) * (legn.deriv()(roots)) ** 2)

        # shift roots and weights from domain [-1, 1] to [0, 1]
        roots = tuple(roots / 2 + 0.5)
        weights = tuple(weights / 2)

        # Calculate absolute positions of Gaussian quadrature roots
        xintegration = [[] for x in range(len(self.nodes) - 1)]
        for i in range(len(self.nodes) - 1):
            for j in range(len(roots)):
                xij = (self.nodes[i+1] - self.nodes[i]) * roots[j] + \
                       self.nodes[i]
                xintegration[i].append(xij)

        self._xgauss = roots
        self._wgauss = weights
        self.gaussquad = [self._xgauss, self._wgauss]
        self.xintegration = xintegration
        self.gauss_degree = degree

    def _aggregate_forcing(self):
        self.forcing = np.repeat(0.0, len(self.nodes))
        # aggregate state independent forcing
        for flux in [self.pointflux, self.spatflux]:
            for key in flux.keys():
                self.forcing += flux[key][-1]

        # add boundaries of type neumann
        for key in self.BCs.keys():
            val, type_, idx = self.BCs[key]
            if type_ == "Neumann":
                self.forcing[idx] += val

    def _internal_forcing(self, calcflux=False, calcbal=False):
        # internal fluxes from previous iteration
        pos, weight = self.gaussquad
        f = [0.0 for x in range(len(self.nodes))]
        for i in range(len(self.nodes) - 1):
            L = self.seg_lengths[i]
            for idx in range(len(pos)):
                x = self.xintegration[i][idx]
                s = self.states[i] * pos[-idx-1] + self.states[i+1] * pos[idx]
                grad = (self.states[i+1] - self.states[i]) / L
                flux = self.systemfluxfunc(x, s, grad)

                if not calcflux:
                    f[i] -= flux * weight[-idx-1]
                f[i+1] += flux * weight[idx]

        if calcflux:
            self.fluxes = np.array(f)
        else:
```

```
            # if balance calculation, don't assign to forcing again
            if not calcbal:
                self.forcing = self.forcing + np.array(f)
            self.internal_forcing["internal_forcing"] = [None, np.array(f)]

    def _statedep_forcing(self):
        # point state dependent forcing
        f = [0.0 for x in range(len(self.nodes))]
        for key in self.Spointflux.keys():
            (Sfunc, (idx_l, idx_r, lfac, rfac)), _ = self.Spointflux[key]
            # calculate state at position by linear interpolation
            dstates = self.states[idx_r] - self.states[idx_l]
            state = self.states[idx_l] + rfac * dstates
            # calculate function value and distribute fluxes accordingly
            value = Sfunc(state)
            f[idx_l] += value * lfac
            f[idx_r] += value * rfac
            self.Spointflux[key][1] = np.array(f)

        # spatial state dependent forcing
        pos, weight = self.gaussquad
        for key in self.Sspatflux.keys():
            f = [0.0 for x in range(len(self.nodes))]
            Sfunc = self.Sspatflux[key][0]
            for i in range(len(self.nodes) - 1):
                L = self.seg_lengths[i]
                for idx in range(len(pos)):
                    x = self.xintegration[i][idx]
                    ds = self.states[i+1] - self.states[i]
                    # linearly interpolate the state value
                    s = self.states[i] + (x - self.nodes[i]) * (ds / L)
                    # assign to nearby nodes according to scheme
                    f[i] += Sfunc(x, s) * weight[-idx-1] * pos[-idx-1] * L
                    f[i+1] += Sfunc(x, s) * weight[idx] * pos[idx] * L

            self.Sspatflux[key][1] = np.array(f)

        # aggregate state dependent forcing
        for flux in [self.Spointflux, self.Sspatflux]:
            for key in flux.keys():
                self.forcing += flux[key][1]

        # add boundaries of type dirichlet
        for key in self.BCs.keys():
            val, type_, idx = self.BCs[key]
            if type_ == "Dirichlet":
                self.forcing[idx] = 0

        # reshape for matrix solver
        self.forcing = np.reshape(self.forcing, (len(self.nodes), 1))

    def _check_boundaries(self):
        # no boundary conditions entered
        keys = list(self.BCs.keys())
        if len(keys) == 0:
            raise np.linalg.LinAlgError("Singular matrix")

        # if one boundary is not entered a zero Neumann boundary is set
        if len(keys) == 1:
            val, type_, pos = self.BCs[keys[0]]
            if type_ == "Dirichlet" and pos == 0:
                self.add_neumann_BC(value=0, where="east")
```

```
            elif type_ == "Dirichlet" and pos == -1:
                self.add_neumann_BC(value=0, where="west")
            else:
                raise np.linalg.LinAlgError("Singular matrix")

        # both boundaries cannot be of type Neumann
        if len(keys) == 2:
            val0, type_0, pos0 = self.BCs[keys[0]]
            val1, type_1, pos1 = self.BCs[keys[1]]
            if type_0 == type_1 and type_0 == "Neumann":
                raise np.linalg.LinAlgError("Singular matrix")

        # constrain the states to the applied boundary conditions
        for key in keys:
            val, type_, pos = self.BCs[key]
            if type_ == "Dirichlet":
                self.states[pos] = val

    def _calc_theta_k(self):
        if hasattr(self, 'kfun'):
            k = [self.kfun(n, s) for n, s in zip(self.nodes, self.states)]
            self.conductivities = np.array(k)
        if hasattr(self, 'tfun'):
            t = [self.tfun(s) for s in self.states]
            self.moisture = np.array(t)

    def _update_storage_change(self, prevstate, dt):
        storagechange = self.Sspatflux.get('storage_change', None)
        if storagechange:
            storagechange[0] = partial(storagechange[0],
                                       prevstate=self.states_to_function(),
                                       dt=dt)

    def _solve_initial_object(self):
        if self.isinitial:
            self._check_boundaries()
            self.forcing = np.repeat(0, len(self.states))
            self._internal_forcing()
            self._update_storage_change(self.states_to_function(), dt=1)

    def _FE_precalc(self):
        slen = np.repeat(0.0, len(self.nodes) - 1)
        nlen = np.repeat(0.0, len(self.nodes))

        # calculate both arrays in one loop & catch boundary cases
        for i in range(len(self.nodes)):
            if i == 0:
                nlen[i] = (self.nodes[i] + self.nodes[i+1]) / 2 - self.nodes[i]
            elif i == len(self.nodes) - 1:
                nlen[i] = self.nodes[i] - (self.nodes[i-1] + self.nodes[i]) / 2
            else:
                nlen[i] = (self.nodes[i+1] - self.nodes[i-1]) / 2

            if i != len(self.nodes) - 1:
                slen[i] = self.nodes[i+1] - self.nodes[i]

        # assign to class attributes
        self.seg_lengths = slen
        self.lengths = nlen

    def _CMAT(self, nodes, states):
        systemflux = self.systemfluxfunc
```

```
        A = np.zeros((len(nodes), len(nodes)))
        # internal flux
        for i in range(len(nodes) - 1):
            # fixed values for selected element
            L = nodes[i+1] - nodes[i]
            stateleft = states[i] + np.array([0, self._delta, 0])
            stateright = states[i+1] + np.array([0, 0, self._delta])
            grad = (stateright - stateleft) / L

            totflux = np.array([0, 0, 0], dtype=np.float64)
            pos, weight = self.gaussquad
            Svall = 0
            Svalr = 0
            # calculates the selected integral approximation
            for idx in range(len(pos)):
                # position and state gaussian integration
                x = self.xintegration[i][idx]
                state_x = stateleft * pos[-idx-1] + stateright * pos[idx]

                flux = [systemflux(x, s, g) for s, g in zip(state_x, grad)]
                totflux += np.array(flux) * weight[idx]

                # calculates gradients for spatial state dependent fluxes
                for key in self.Sspatflux.keys():
                    Sfunc = self.Sspatflux[key][0]
                    Sval = [Sfunc(x, s) for s in state_x]
                    dsl = (Sval[1] - Sval[0]) / self._delta
                    dsr = (Sval[2] - Sval[0]) / self._delta

                    # distribution of flux according to gaussian integration
                    Svall += pos[-idx-1] * weight[-idx-1] * L * dsl
                    Svalr += pos[idx] * weight[idx] * L * dsr

            dfluxl = (totflux[1] - totflux[0]) / self._delta
            dfluxr = (totflux[2] - totflux[0]) / self._delta

            # assign flux values to the coefficient matrix
            A[i][i] += -dfluxl + Svall
            A[i+1][i] += dfluxl + Svall
            A[i][i+1] += -dfluxr + Svalr
            A[i+1][i+1] += dfluxr + Svalr

        # state dependent point flux
        for key in self.Spointflux.keys():
            (Sfunc, (idx_l, idx_r, lfac, rfac)), _ = self.Spointflux[key]
            # calculate state at position by linear interpolation
            dstates = self.states[idx_r] - self.states[idx_l]
            state = self.states[idx_l] + rfac * dstates
            sfunc_s = Sfunc(state)
            sfunc_sd = Sfunc(state + self._delta)
            dfunc = (sfunc_sd - sfunc_s) / self._delta
            A[idx_l][idx_l] += dfunc * lfac
            A[idx_r][idx_r] += dfunc * rfac

        self.coefmatr = A

    def set_field1d(self, nodes, degree=1):
        if isinstance(nodes, tuple):
            self.nodes = np.linspace(*nodes)
        else:
            self.nodes = np.array(nodes)

```

```
        self.states = np.repeat(0.0, len(self.nodes))
        self.forcing = np.repeat(0.0, len(self.nodes))
        self.set_gaussian_quadrature(degree=degree)
        self._FE_precalc()

    def set_systemfluxfunction(self, function, **kwargs):
        for k, v in kwargs.items():
            setattr(self, k, v)

        def fluxfunction(x, s, gradient):
            return function(x, s, gradient, **kwargs)
        self.systemfluxfunc = fluxfunction

    def set_initial_states(self, states):
        if isinstance(states, (int, float)):
            states = float(states)
            self.states = np.array([states for x in range(len(self.nodes))])
        else:
            self.states = np.array(states, dtype=np.float64)

    def add_dirichlet_BC(self, value, where):
        if isinstance(value, int) or isinstance(value, float):
            value = [value]
            where = [where]

        for val, pos in zip(value, where):
            if pos.lower() in ["west", "left", "down"]:
                self.BCs["west"] = (val, "Dirichlet", 0)
                self._west = 1
            elif pos.lower() in ["east", "right", "up"]:
                self.BCs["east"] = (val, "Dirichlet", -1)
                self._east = -1

    def add_neumann_BC(self, value, where):
        if isinstance(value, int) or isinstance(value, float):
            value = [value]
            where = [where]

        for val, pos in zip(value, where):
            if pos.lower() in ["west", "left", "down"]:
                self.BCs["west"] = (val, "Neumann", 0)
                self._west = 0
            elif pos.lower() in ["east", "right", "up"]:
                self.BCs["east"] = (val, "Neumann", -1)
                self._east = len(self.nodes)

    def remove_BC(self, *args):
        if len(args) == 0:
            self.BCs = {}
            self._west = None
            self._east = None
        else:
            for name in args:
                try:
                    self.BCs.pop(name)
                    if name == "west":
                        self._west = None
                    else:
                        self._east = None
                except KeyError as e:
                    raise type(e)("No boundary named " + str(name) + ".")

```

```
433     def add_pointflux(self, rate, pos, name=None):
434         f = [x*0.0 for x in range(len(self.nodes))]
435         if isinstance(rate, int) or isinstance(rate, float):
436             rate = [rate]
437             pos = [pos]
438 
439         # add single valued pointflux to corresponding control volume
440         if isinstance(rate, list):
441             for r, p in zip(rate, pos):
442                 # find indices left and right of pointflux
443                 idx_r = np.searchsorted(self.nodes, p)
444                 idx_l = idx_r - 1
445                 # calculate contribution of pointflux to neighbouring nodes
446                 nodedist = self.nodes[idx_r] - self.nodes[idx_l]
447                 lfactor = 1 - (p - self.nodes[idx_l]) / nodedist
448                 rfactor = 1 - lfactor
449                 # assign to the forcing vector
450                 f[idx_l] += r * lfactor
451                 f[idx_r] += r * rfactor
452 
453             if name:
454                 self.pointflux[name] = [np.array(f)]
455             else:
456                 name = str(Time.time())
457                 self.pointflux[name] = [np.array(f)]
458 
459         # add state dependent pointflux
460         elif callable(rate):
461             # find indices left and right of pointflux
462             idx_r = np.searchsorted(self.nodes, pos)
463             idx_l = idx_r - 1
464             # calculate contribution of pointflux to neighbouring nodes
465             nodedist = self.nodes[idx_r] - self.nodes[idx_l]
466             lfactor = 1 - (pos - self.nodes[idx_l]) / nodedist
467             rfactor = 1 - lfactor
468 
469             if not name:
470                 name = rate.__name__
471 
472             # create data structure
473             self.Spointflux[name] = [(rate, (idx_l, idx_r, lfactor, rfactor)),
474                                      np.array(f)]
475 
476     def add_spatialflux(self, q, name=None):
477         if isinstance(q, int) or isinstance(q, float):
478             Q = np.repeat(q, len(self.nodes)).astype(np.float64)
479         elif isinstance(q, list) or isinstance(q, np.ndarray):
480             Q = np.array(q).astype(np.float64)
481         elif callable(q):
482             Q = q
483 
484         if not callable(Q):
485             f = Q * self.lengths
486 
487             if not name:
488                 name = str(Time.time())
489 
490             self.spatflux[name] = [np.array(f)]
491 
492         else:
493             # prepare a domain spaced and zero-ed array
494             f = [x*0.0 for x in range(len(self.nodes))]
```

```
495
496             # check callable's number of positional arguments
497             fparams = signature(Q).parameters
498             nargs = sum(i == str(j) for i, j in fparams.items())
499
500             if not name:
501                 name = Q.__name__
502
503             # if function has one argument > f(x)
504             if nargs == 1:
505                 pos, weight = self.gaussquad
506                 for i in range(len(self.nodes) - 1):
507                     # distance between nodes
508                     L = self.seg_lengths[i]
509                     for idx in range(len(pos)):
510                         x = self.xintegration[i][idx]
511                         # to left node (no pos negatives?)
512                         f[i] += Q(x) * weight[-idx-1] * pos[-idx-1] * L
513                         # to right node
514                         f[i+1] += Q(x) * weight[idx] * pos[idx] * L
515                 self.spatflux[name] = [np.array(f)]
516
517             # if function has two arguments > f(x,s)
518             elif nargs == 2:
519                 self.Sspatflux[name] = [Q, np.array(f)]
520
521             elif nargs == 3:
522                 # implement time dependence ??
523                 pass
524
525             # only valid for the storage change function
526             # f(x, s, prevstate, dt)
527             elif nargs == 4:
528                 name = 'storage_change'
529                 self.Sspatflux[name] = [Q, np.array(f)]
530
531     def remove_pointflux(self, *args):
532         # remove all point fluxes
533         if len(args) == 0:
534             self.pointflux = {}
535             self.Spointflux = {}
536         else:
537             # remove the specific pointflux name(s)
538             for name in args:
539                 try:
540                     self.pointflux.pop(name)
541                 except KeyError:
542                     try:
543                         self.Spointflux.pop(name)
544                     except KeyError as e:
545                         # Only raise exception at top of call stack
546                         raise KeyError('{} is not a pointflux.'.format(e)) from
    None
547
548     def remove_spatialflux(self, *args):
549         # remove all spatial fluxes when args is empty
550         if len(args) == 0:
551             self.spatflux = {}
552             self.Sspatflux = {}
553         else:
554             # remove the given spatial flux name(s)
555             for name in args:
```

```
556                 try:
557                     self.spatflux.pop(name)
558                 except KeyError:
559                     try:
560                         self.Sspatflux.pop(name)
561                     except KeyError as e:
562                         # Only raise exception at top of call stack
563                         raise KeyError('{} is not a spatialflux.'.format(e)) from
    None
564
565     def states_to_function(self):
566         states = self.states.copy()
567         # check if west boundary is of type Dirichlet
568         if self._west == 1:
569             states[0] = self.BCs["west"][0]
570         # check is east boundary is of type Dirichlet
571         if self._east == -1:
572             states[-1] = self.BCs["east"][0]
573         # linearly interpolate between states including assigned boundaries
574         return partial(np.interp, xp=self.nodes, fp=states)
575
576     def dt_solve(self, dt, maxiter=500, threshold=1e-3, store=False):
577         self._check_boundaries()
578         west = self._west
579         east = self._east
580
581         # if system is transient
582         self._update_storage_change(self.states_to_function(), dt)
583
584         itercount = 1
585         while itercount <= maxiter:
586             self._aggregate_forcing()
587             self._internal_forcing()
588             self._statedep_forcing()
589             self._CMAT(self.nodes, self.states)
590
591             solution = np.linalg.solve(self.coefmatr[west:east, west:east],
592                                        -1*self.forcing[west:east]).flatten()
593
594             prevstates = self.states
595             curstates = np.copy(prevstates)
596             curstates[west:east] += solution
597             self.states = curstates
598
599             # update forcing and boundaries at new states
600             self._aggregate_forcing()
601             self._internal_forcing()
602             self._statedep_forcing()
603
604             if store:
605                 self.nr_states.append(self.states)
606
607             # check local/inner loop solution for conversion
608             if converged(prevstates, curstates, threshold):
609                 break
610
611             itercount += 1
612         else:
613             return itercount
614         self.isinitial = False
615         return itercount
616
```

```
617     def solve(self, dt=0.001, dt_min=1e-5, dt_max=0.5, end_time=1, maxiter=500,
618               dtitlow=1.5, dtithigh=0.5, itermin=5, itermax=10, threshold=1e-3,
619               global_threshold=0.0, verbosity=True, storeNR=False):
620         solved_objs = [deepcopy(self)]
621         time_data = [0]
622         dt_data = [None]
623         iter_data = [None]
624 
625         time = dt
626 
627         t0 = Time.clock()
628         while time <= end_time:
629             # solve for given dt
630             iters = self.dt_solve(dt, maxiter, threshold, store=storeNR)
631 
632             # catch cases where maxiter is reached
633             if iters > maxiter:
634                 if dt == dt_min:
635                     print(f'Maxiter {iters} at dt_min {dt_min} reached')
636                     self.isconverged = False
637                     break
638                 else:
639                     print(f'Maxiter {iters} reached, dt {dt} is lowered...')
640                     dt *= dtithigh
641                     if dt < dt_min:
642                         dt = dt_min
643                     # revert back to previous model state
644                     self.states = solved_objs[-1].states
645                     continue
646 
647             self.isconverged = True
648 
649             if verbosity:
650                 if self.Sspatflux.get('storage_change', None):
651                     fmt = 'Converged at time={} for dt={} with {} iterations'
652                     print(fmt.format(time, dt, iters))
653                 else:
654                     print('Converged at {} iterations'.format(iters))
655 
656             # break out of loop when system is stationary
657             if not self.Sspatflux.get('storage_change', None):
658                 solved_objs.append(deepcopy(self))
659                 time_data.append(time)
660                 dt_data.append(dt)
661                 iter_data.append(iters)
662                 break
663 
664             # build data record
665             solved_objs.append(deepcopy(self))
666             time_data.append(time)
667             dt_data.append(dt)
668             iter_data.append(iters)
669 
670             # check for global/outer loop convergence
671             if converged(solved_objs[-2].states, solved_objs[-1].states,
672                          global_threshold):
673                 break
674 
675             # adapt dt as function of iterations of previous step
676             if iters <= itermin:
677                 dt *= dtitlow
678                 if dt > dt_max:
```

```
                    dt = dt_max
            elif iters >= itermax:
                dt *= dtithigh
                if dt < dt_min:
                    dt = dt_min

            # last time step calculated should be end_time exactly
            remaining_time = end_time - time
            if remaining_time == 0:
                pass
            elif remaining_time < dt:
                dt = remaining_time

            # increment time
            time += dt

        self.runtime = Time.clock() - t0

        # attach all solve data to last created object
        solve_data = {}
        solve_data['solved_objects'] = solved_objs
        solve_data['time'] = time_data
        solve_data['dt'] = dt_data
        solve_data['iter'] = iter_data
        self.solve_data = solve_data

    def calcbalance(self, print_=False, invert=True):
        data = {}
        # internal fluxes
        self._internal_forcing(calcbal=True)
        self._internal_forcing(calcflux=True)
        internalfluxes = self.internal_forcing['internal_forcing'][-1]

        # add all point fluxes
        pnt = np.repeat(0.0, len(self.nodes))
        pointfluxes = {**self.pointflux, **self.Spointflux}
        for key in pointfluxes.keys():
            # if state dependent, calculate new forcing for new states
            if key in self.Spointflux.keys():
                f = [0.0 for x in range(len(self.nodes))]
                (Sfunc, (idx_l, idx_r, lfac, rfac)), _ = self.Spointflux[key]
                # calculate state at position by linear interpolation
                dstates = self.states[idx_r] - self.states[idx_l]
                state = self.states[idx_l] + rfac * dstates
                # calculate function value and distribute fluxes accordingly
                value = Sfunc(state)
                f[idx_l] += value * lfac
                f[idx_r] += value * rfac
                pnt += np.array(f)
            # if constant/position dependent then no need for a new calculation
            else:
                pnt += pointfluxes[key][-1]
            data.update({"pnt-"+str(key): pointfluxes[key][-1]})

        # add all spatial fluxes
        spat = np.repeat(0.0, len(self.nodes))
        spatialfluxes = {**self.spatflux, **self.Sspatflux}
        for key in spatialfluxes.keys():
            # if state dependent, calculate new forcing for new states
            if key in self.Sspatflux.keys():
                pos, weight = self.gaussquad
                f = [0.0 for x in range(len(self.nodes))]
```

```
                Sfunc = self.Sspatflux[key][0]
                for i in range(len(self.nodes) - 1):
                    L = self.seg_lengths[i]
                    for idx in range(len(pos)):
                        x = self.xintegration[i][idx]
                        ds = self.states[i+1] - self.states[i]
                        # linearly interpolate the state value
                        s = self.states[i] + (x - self.nodes[i]) * (ds / L)
                        # assign to nearby nodes according to scheme
                        f[i] += Sfunc(x, s) * weight[-idx-1] * pos[-idx-1] * L
                        f[i+1] += Sfunc(x, s) * weight[idx] * pos[idx] * L
                spat += np.array(f)
            # if constant/position dependent then no need for a new calculation
            else:
                spat += spatialfluxes[key][-1]
            data.update({"spat-"+str(key): spatialfluxes[key][-1]})

        # storage between iterations, if exists
        try:
            storage_change = data.pop("spat-storage_change")
        except KeyError:
            storage_change = np.repeat(0.0, len(self.nodes))

        # remove storage change from external spatial forcings
        spat = spat - storage_change

        # internal balance
        internalfluxes = internalfluxes + pnt + spat

        # boundaries
        lbound = internalfluxes[0] + storage_change[0]
        rbound = internalfluxes[-1] + storage_change[-1]

        # correct for boundary fluxes
        internalfluxes[0] -= lbound
        internalfluxes[-1] -= rbound
        self.fluxes[0] -= lbound

        # net flow
        net = internalfluxes + storage_change

        # dump waterbalance & summary to dataframe
        data.update({'storage_change': storage_change,
                     'internal': internalfluxes, 'all-spatial': spat,
                     'all-points': pnt, 'all-external': pnt + spat,
                     'net': net, 'fluxes': self.fluxes})

        df_balance = pd.DataFrame(data)
        df_balance.insert(0, 'nodes', self.nodes)

        if invert:
            df_balance = df_balance.iloc[::-1].reset_index(drop=True)

        self.df_balance = df_balance
        df_balance_summary = df_balance.sum().transpose().drop(['nodes', 'fluxes'
    ])
        self.df_balance_summary = df_balance_summary

        if print_:
            print(self.df_balance_summary)

    def dataframeify(self, invert):
```

```
        self._calc_theta_k()
        self._solve_initial_object()

        columns = ['lengths', 'nodes', 'states', 'moisture',
                   'conductivities', 'pointflux', 'Spointflux',
                   'spatflux', 'Sspatflux', 'internal_forcing']
        data = {}
        for idx, name in enumerate(columns):
            if 0 <= idx < 3:
                data[name] = self.__dict__.get(name)
            elif 3 <= idx < 5:
                values = self.__dict__.get(name)
                if list(values):
                    data[name] = values
            else:
                for k, v in self.__dict__.get(name).items():
                    data[k] = v[-1]

        df = pd.DataFrame(data)
        if invert:
            df = df.iloc[::-1].reset_index(drop=True)
        self.df_states = df

    def transient_dataframeify(self, print_times=None, include_maxima=True,
                               nodes=None, invert=True):
        # times at which the model has been solved
        self.dft_solved_times = pd.DataFrame(data=self.solve_data)

        # solve model at specific print times
        if print_times:
            st = self.solve_data['time']
            solved_obj = self.solve_data['solved_objects']

            # print_times can be a sequence or a scalar
            if isinstance(print_times, (list, np.ndarray)):
                pt = np.array(print_times)
                pt = np.delete(pt, np.argwhere(pt < min(st)))
                pt = np.delete(pt, np.argwhere(pt > max(st)))
                if include_maxima:
                    pt = np.insert(pt, 0, min(st))
                    pt = np.append(pt, max(st))
            else:
                pt = np.linspace(min(st), max(st), print_times)
            pt.sort()
            pt = np.unique(pt)

            # Find solved model that is nearest in terms of time
            # solve the model for the new time from here
            new_obj = []
            for t, i in zip(pt, np.searchsorted(st, pt)):
                # print time already equals known state
                if t == st[i]:
                    new_obj.append(deepcopy(solved_obj[i]))
                # calculate model state at new time
                else:
                    dts = t - st[i - 1]
                    obj = deepcopy(solved_obj[i - 1])
                    obj.dt_solve(dts)
                    new_obj.append(obj)

            # dump model states at print times to dataframe
            data = {'solved_objects': new_obj, 'time': pt}
```

```
            dft_print_times = pd.DataFrame(data=data)
            self.dft_print_times = dft_print_times

        # if no specific print times remove dataframe
        else:
            self.dft_print_times = None

        # if available, use print times instead of solved_times
        if self.dft_print_times is not None:
            timedf = self.dft_print_times
        else:
            timedf = self.dft_solved_times

        # build all dataframes
        dft_states = {}
        dft_balance = {}
        dft_nodes = {}
        dft_bal_sum = pd.DataFrame()
        for row in timedf.itertuples(index=False):
            obj, t = row.solved_objects, row.time

            # states dataframe
            obj.dataframeify(invert=invert)
            dft_states.update({t: obj.df_states})

            # balance dataframes
            obj.calcbalance(invert=invert)
            dft_balance.update({t: obj.df_balance})
            dft_bal_sum = dft_bal_sum.append(obj.df_balance_summary,
                                             ignore_index=True)

            # track specific nodes (if present)
            if nodes:
                if not dft_nodes:
                    dft_nodes = dict((k, np.array([])) for k in nodes)
                df = obj.df_states
                c = df.columns
                for node in nodes:
                    nrow = df[df['nodes'] == node].to_numpy()
                    if len(dft_nodes[node]):
                        dft_nodes[node] = np.vstack((dft_nodes[node], nrow))
                    else:
                        dft_nodes[node] = nrow

        # nodes dataframe (if present)
        if nodes:
            d = {}
            for k, v in dft_nodes.items():
                d[k] = pd.DataFrame(v, columns=c)
                d[k].insert(0, timedf.time.name, timedf.time)
            self.dft_nodes = d

        self.dft_states = dft_states
        self.dft_balance = dft_balance
        dft_bal_sum.insert(0, timedf.time.name, timedf.time)
        self.dft_balance_summary = dft_bal_sum

    def save(self, savepath=None, dirname=None):
        savepath = savepath or self.savepath
        if not os.path.isdir(savepath):
            os.mkdir(savepath)

```

```
        # save directory
        if not dirname:
            # chronological name
            dirname = Time.strftime('%d%b%Y_%H%M%S', Time.gmtime(Time.time()))
            runpath = os.path.join(savepath, dirname)
            os.mkdir(runpath)
        else:
            # given name
            runpath = os.path.join(savepath, dirname)
            if not os.path.isdir(runpath):
                os.mkdir(runpath)

        # write transient dataframes
        for k, v in self.__dict__.items():
            # transient dataframes only
            if k.startswith('dft_'):
                fname = f"{k}.xlsx"
                save_file = os.path.join(runpath, fname)

                if isinstance(v, pd.core.frame.DataFrame):
                    with pd.ExcelWriter(save_file) as fw:
                        v.to_excel(fw, sheet_name=k)

                elif isinstance(v, dict):
                    with pd.ExcelWriter(save_file) as fw:
                        for k, df in v.items():
                            df.to_excel(fw, sheet_name=str(k))

        # write model summary
        self.summary(show=False, save=True, path=runpath)


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Code 2: One-dimensional flow model.

```
import os
from collections import namedtuple

import pandas as pd

from waterflow import DATA_DIR


def soilselector(soils):
    staringreeks = pd.read_table(os.path.join(DATA_DIR, "StaringReeks.txt"),
                                 delimiter="\t")
    soildata = staringreeks.iloc[[s-1 for s in soils]]

    # useful for a plotting domain
    minima = namedtuple('min', staringreeks.columns)(*soildata.min())
    maxima = namedtuple('max', staringreeks.columns)(*soildata.max())
    extrema = (minima, maxima)

    # turn row(s) to namedtuple
    rows = list(soildata.itertuples(name='soil', index=False))
    return rows, soildata, extrema


def VG_theta(theta, theta_r, theta_s, a, n):
    m = 1-1/n
    THETA = (theta_s - theta_r) / (theta - theta_r)
    return ((THETA**(1/m) - 1) / a**n)**(1/n)


def VG_pressureh(h, theta_r, theta_s, a, n):
    # to theta
    if h >= 0:
        return theta_s
    m = 1-1/n
    return theta_r + (theta_s-theta_r) / (1+(a*-h)**n)**m


def VG_conductivity(x, h, ksat, a, n):
    if h >= 0:
        return ksat
    m = 1-1/n
    h_up = (1 - (a * -h)**(n-1) * (1 + (a * -h)**n)**-m)**2
    h_down = (1 + (a * -h)**n)**(m / 2)
    return (h_up / h_down) * ksat


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Code 3: Conductivity functions.

```
def darcy(x, s, gradient, kfun=lambda x, s: 1):
    return - kfun(x, s) * gradient


def darcy_s(x, s, gradient, kfun=lambda x, s: 1):
    return - kfun(x, s) * s * gradient


def richards_equation(x, s, gradient, kfun):
    return -kfun(x, s) * (gradient + 1)


def storage_change(x, s, prevstate, dt, fun=lambda x: x, S=1.0):
    return - S * (fun(s) - fun(prevstate(x))) / dt


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Code 4: Flux functions.

```python
import numpy as np


def spacing(nx, Lx, ny=0, Ly=0, linear=True, loc=None, power=None, weight=None):
    axes_args = np.linspace(0, Lx, nx), np.linspace(0, Ly, ny)
    if linear:
        return axes_args[0], axes_args[1]
    else:
        # check dimensions
        loc = np.array(loc)
        if len(np.shape(loc)) == 1:
            xloc = loc[np.argsort(loc)]
            yloc = np.empty(0)
        if len(np.shape(loc)) == 2:
            xloc = np.unique(loc[:, 0][np.argsort(loc[:, 0])])
            yloc = np.unique(loc[:, 1][np.argsort(loc[:, 1])])
        # start populating the axes
        axes = np.array([np.repeat(0.0, nx), np.repeat(0.0, ny)])
        for iloc, loc in enumerate([xloc, yloc]):
            # dimension is not calculated if axis is empty
            if len(loc) == 0:
                break
            # select new axis
            ax_arg = axes_args[iloc]
            axis = axes[iloc]
            # per positions on the axis
            for pts_i in range(len(loc)):
                p = loc[pts_i]
                # add the center starting point
                axis[p] = axes_args[iloc][p]
                # continue with populating the other positions in range power
                for i in range(1, power + 1):
                    # to right
                    axis[p+i] = axis[p+i-1] + (ax_arg[p+i] - axis[p+i-1]) / weight
                    # to left
                    axis[p-i] = axis[p-i+1] - (axis[p-i+1] - ax_arg[p-i]) / weight
                # fill axis left of the point to zero with linear distance
                if pts_i == 0:
                    fill_left = np.linspace(0, axis[p-power], p - power + 1)
                    axis[:p-power+1] = fill_left
                # fill axis left to previous point with linear distance
                else:
                    fill_left = np.linspace(axis[loc[pts_i-1]+power],
                                            axis[p-power],
                                            (p-power + 1) - (loc[pts_i-1] + power)
    )
                    axis[loc[pts_i-1]+power:p-power+1] = fill_left
            # fill axis right of the point with linear distance
            fill_right = np.linspace(axis[p+power],
                                     ax_arg[-1],
                                     len(axis) - (p+power))
            axis[p+power:] = fill_right
            # reassign axis at which spacing is completed
            axes[iloc] = axis
    return axes[0], axes[1]


def biasedspacing(numnodes, power, lb=0, rb=1, maxdist=None, length=1):
    # at least two nodes are needed to define a domain
    if numnodes <= 2:
        return np.array([lb, rb]) * length
    # equal spacing
```

```
    if power <= 1:
        return np.linspace(lb, rb, numnodes) * length

    arr = [lb]
    # build discretization iteratively
    for n in range(numnodes - 2, 0, -1):
        i = (rb - lb) / (power * n)
        arr.append(i + arr[-1])
        lb = arr[-1]
    arr.append(rb)
    arr = np.array(arr) * length

    # if maxdist is exceeded, shift nodes proportionally
    if maxdist:
        sign = np.sign(rb - lb)
        fraction_prev = 0
        for i in range(numnodes - 2):
            idxl, idxr = numnodes - i - 2, numnodes - i - 1
            dist = abs(arr[idxr] - arr[idxl])
            fraction = (dist - maxdist) / dist
            if fraction >= 0:
                fraction_prev = fraction
                arr[idxl] += fraction * dist * sign
            else:
                diff = arr[2:idxr+1] - arr[1:idxr]
                arr[1:idxr] += diff * fraction_prev
                break
    return arr


if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Code 5: Discretization functions.

```
import os
from functools import partial, update_wrapper


def converged(old_states, new_states, threshold):
    max_abs_change = max(abs(old_states - new_states))
    max_abs_allowed_change = max(abs(threshold * new_states))

    return max_abs_change < max_abs_allowed_change


def initializer(func, *args, **kwargs):
    pfunc = partial(func, *args, **kwargs)
    # update initialized function with function's original attributes
    update_wrapper(pfunc, func)
    return pfunc


def newdir(basepath, dirname):
    newpath = os.path.join(basepath, dirname)
    if not os.path.isdir(newpath):
        os.mkdir(newpath)
    return newpath


if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

Code 6: Helper functions.