# Contents



# 1 Introduction

This model couples two pre-existing models: a 2D SVE solver [*Bradford and Katopodes*, 1999] and a 1D Richards equation solver [*Celia et al.*, 1990]. The SVE solver uses a finite volume method involving two steps, predictor and corrector, to achieve second order accuracy. The SVE model is coupled at each grid cell and timestep to the Richards equation solver. The core of the model is implemented in Fortran (`dryR.for`), and Python scripts are provided to write and read the Fortran files.

This document is divided into the following sections:

- Section 2: A brief summary of the Saint Venant and Richards Equations.
- Section 3: An overview the SVE-R numerical methods, and how the two model components are coupled.
- Section 3: An overview of the Python wrapper scripts, which write the Fortran input files, compile and execute the Fortran code, and read and visualize the Fortran outputs.
- Additional Python code: a stand-alone Richards solver
- Additional Python code: an implementation of **?**.

# 2 Background: Equations and Numerical Methods

## 2.1 The 2D Saint Venant Equations

The SVE are written in integral form:

$$\frac{\partial}{\partial t}\int_{\Omega}\mathbf{U}d\Omega + \oint_{\partial}(\mathbf{F}dy - \mathbf{G}dx) = \int_{\Omega}\mathbf{Q}d\Omega \tag{1}$$

where $\mathbf{U}^T = (h, hU, hV)$ is the vector of conservative variables.

$$\mathbf{F} = \begin{bmatrix} hU \\ hU^2 + \frac{1}{2}gh^2 \\ hUV \end{bmatrix}; \quad \mathbf{G} = \begin{bmatrix} hV \\ hUV \\ hV^2 + \frac{1}{2}gh^2 \end{bmatrix}$$

where $h$ is the flow depth, $z$ is the bed elevation, $u$ and $v$ are the vertically averaged velocities in the $x$ and $y$ directions, respectively. The source terms are defined as:

$$\mathbf{Q} = \begin{bmatrix} p - i \\ -gh\dfrac{\partial z}{\partial x} - ghS_{f,x} + \dfrac{u(p-i)}{2} \\ -gh\dfrac{\partial z}{\partial y} - ghS_{f,y} + \dfrac{v(p-i)}{2} \end{bmatrix}$$

where $p$ is the rainfall rate, $i$ is the infiltration rate of water into the bed, and $S_{f,x}$ and $S_{f,y}$ are the $x$ and $y$ components of the friction slope.

## 2.2 SVE numerical methods

The model is adapted from *Bradford and Katopodes* (1999), referred to here as BK, and uses predictor-corrector time-stepping to provide a second-order accurate solution. The only significant difference

between the BK model and the model described here is the coupling to a Richards equation solver. The following summary of the numerical methods is adapted from *Bradford and Katopodes* (1999), where a better and more complete explanation can be found.

A predictor is computed at time level $n + 1/2$ by solving the primitive equations in generalized coordinates, and a corrector is computed at the $n + 1$ time level by solving the integral equations.

### Predictor Step

The predictor solution is computed by solving the equations in primitive form. In generalized coordinates:

$$\frac{\partial \mathbf{W}}{\partial t} + \mathbf{A}_W \frac{\partial \mathbf{W}}{\partial \xi} + \mathbf{B}_W \frac{\partial \mathbf{W}}{\partial \eta} = \mathbf{Q}_W \tag{2}$$

where $\xi$ and $\eta$ are in the directions of increasing $j$ and $k$ indices, respectively. The $j$,$k$ indices indicate the column and row numbers of a given cell, respectively (see Figure 1 schematic). $\mathbf{W_T} = [h, U, V]$ is the array of primitive variables, and the matrices $\mathbf{A}_W$ and $\mathbf{B}_W$ are defined as:

$$\mathbf{A}_w = \begin{bmatrix} U_\xi & h\xi_x & h\xi_y \\ g\xi_x & U_\xi & 0 \\ g\xi_y & 0 & U_\xi \end{bmatrix} \quad \mathbf{B}_w = \begin{bmatrix} U_\eta & h\eta_x & h\eta_y \\ g\eta_x & U_\eta & 0 \\ g\eta_y & 0 & U_\eta \end{bmatrix}$$

where $U_\xi = U\xi_x + V\xi_y$ and $U_\eta = U\eta_x + V\eta_y$. $\xi_x$, $\xi_y$, $\eta_x$ and $\eta_y$ are the grid transformation metrics for mapping $x$ and $y$ to $\xi$ and $\eta$. In Cartesian coordinates, the generalized coordinates simplify to: $\xi = x$ and $\eta = y$.

The predictor solution in cell $j$, $k$ at $t + \Delta t/2$ is given as:

$$\mathbf{W}_{j,k}^{n+1/2} = \mathbf{W}_{j,k}^{n} - \frac{\Delta t}{2}(\mathbf{A}_W \overline{\Delta \mathbf{W}}_\xi + \mathbf{B}_W \overline{\Delta \mathbf{W}}_\eta - \mathbf{Q}_W)_{j,k}^{n+1/2} \tag{3}$$

where the overbar denotes a cell-average gradient of $\mathbf{W}$ in cell $j$, $k$, which is computed with a flux limiter (nonlinear average) in order to preserve solution monotonicity. Flux limiters become first-order accurate near discontinuities while remaining second-order accurate elsewhere, and several options are included in the code, described in Section 3.

### Corrector Step

The corrector solution is obtained from the conservative form of the governing equations.

The predictor solutions are reconstructed to the left and right of each cell face using the monotone upstream scheme for conservation laws (MUSCL), which achieves second-order spatial accuracy. The

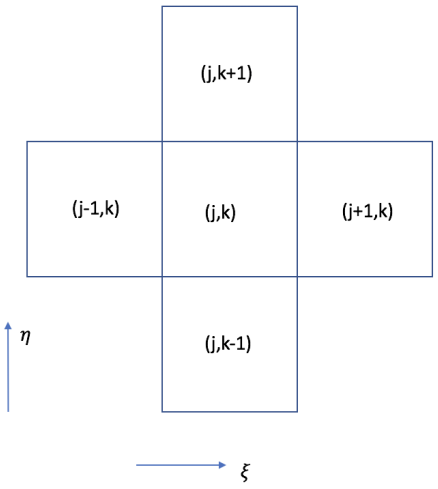


Figure 1: Sketch of a computational cell, after BK Figure 2.

reconstructed predictor values define a Reimann problem at each cell face, which are used to compute the interfacial fluxes:

$$\frac{\mathbf{U}_{j,k}^{n+1} - \mathbf{U}_{j,k}^{n}}{\Delta t} + \frac{1}{\Omega_{j,k}}\big[ - \mathbf{F}_{\perp 1}^{n+1/2}\Delta s_1 + \mathbf{F}_{\perp 2}^{n+1/2}\Delta s_2 + \mathbf{F}_{\perp 3}^{n+1/2}\Delta s_3 - \mathbf{F}_{\perp 4}^{n+1/2}\Delta s_4\big] = \mathbf{Q}^{n+1/2}$$

where $\mathbf{Q}$ and $\mathbf{U}$ are the cell-center values in cell $j, k$ with area $\Omega$, $\mathbf{F}$ and $\mathbf{G}$ are the average boundary values on each cell face, and $\delta s$ is the length of the cell faces. The indices 1 through 4 denote the four cell faces: index 1 corresponds to the bottom cell face, and the remaining cell faces are numbered in counter-clockwise order.

The flux $\mathbf{F}$ is normal to the cell boundary and positive in the direction of increasing cell coordinates. $\mathbf{F}$ is defined as:

$$\mathbf{F}_{\perp} = \begin{bmatrix} hu_{\perp} \\ huu_{\perp} + \frac{1}{2}gh^2\cos\phi \\ hvu_{\perp} + \frac{1}{2}gh^2\sin\phi \end{bmatrix}$$

where $u_{\perp}$ is the velocity perpendicular to the cell face, and $\phi$ is the angle between the face normal vector and the $x$-axis.

The fluxes are evaluated using a Godunov-type upwind scheme in which a Riemann problem is solved across each cell face, using the method of Roe (1981). Further details can be found in BK.

**The source term**

The source term $\mathbf{Q}$ contains the parameterization of the surface roughness (via the friction slope $S_f$ and the lateral inputs (rainfall $p$ and infiltration $i$). Infiltration is independently modeled at each timestep and grid cell with the 1D Richards equation, described in the next subsection.

The friction slope is specified in the generalized form:

$$S_{f,x} = \left(\frac{\alpha U}{h^m}\right)^{1/\eta} \frac{|U|}{U}; \quad S_{f,y} = \left(\frac{\alpha V}{h^m}\right)^{1/\eta} \frac{|U|}{V}$$

where $\alpha$ is a roughness parameter, and $m$ specifies the flow regime ($m = 2$ for laminar flow, 1/2 for turbulent flow). $\eta$=1/2 for most roughness schemes, with the exception of laminar flow, for which $\eta$=1. For Manning's equation, $\alpha = n$, $m = 2/3$ and $\eta = 1/2$:

$$S_{f,x} = \frac{n^2 U}{h^{4/3}}|U|; \quad S_{f,y} = \frac{n^2 V}{h^{4/3}}|U|$$

More generally, with $\eta = 1/2$:

$$S_{f,x} = \frac{\alpha^2}{h^m} U|U|; \quad S_{f,y} = \frac{\alpha^2}{h^m} V|U|$$

**Richards equation**

Richards equation is solved following the approach outlined by *Celia et al.* (1990), which involves a backward Euler approximation in time coupled with a simple Picard iteration scheme. The solver used the discrete approximation of the mixed $H$-$\theta$ form:

$$\frac{\partial \theta}{\partial t} - \nabla \cdot K \nabla H - \frac{\partial K}{\partial z} = 0 \tag{4}$$

where $z$ denotes the vertical dimension (assumed positive upwards), $\theta$ is the soil moisture content, $H$ is the matric potential and $K$ is the unsaturated hydraulic conductivity. $\theta$ and matric potential $H$ are related via the Van Genuchten water retention curve. The volumetric soil moisture content, $\theta$, and effective saturation, $S_e$, are computed as:

$$\theta = \frac{\theta_S – \theta_R}{1 + (\alpha|H|)^n)^m} + \theta_R$$

$$S_e = \frac{\theta – \theta_R}{\theta_S – \theta_R}$$

where $\theta_S$ and $\theta_R$ are the saturated and residual soil moisture content; $n$ is a measure of the pore size distribution;$m = (1 - 1/n)$; and $\alpha$ is related to the inverse of the air entry suction. The unhydraulic conductivity $K$ is computed as:

$$K = K_s \sqrt{S_e}[1–(1–S_e^{1/m})^m]^2$$

where $K_s$ is the saturated hydraulic conductivity. The infiltration rate is solved with Darcy's law:

$$q = -K\left(\frac{\partial H}{\partial z} + 1\right) \tag{5}$$

where the 1 (second term) on the RHS of Equation 5 reflects the fact that $H$ is the matric head, as opposed to the hydraulic head.

### Coupling the SVE and Richards Equation models

The model components are coupled at each grid cell in two steps: the depth from the SVE solver provides the surface boundary condition to the Richards equation solver, and the infiltration rate from the Richards equation solver is used by the SVE source term. This requires that several cases be accounted for: (1) no rain and no ponding, (2) rain but no ponding, and (3) ponding (with or without rain). In case (1), a no flux boundary condition is applied at the surface. In case (2), the Richards equation solver computes a potential infiltration rate ($PI$), defined as the infiltration rate that would occur with $H = 0$ cm at the surface, and compares this value to the rainfall intensity, $p$. If $p$ exceeds the potential infiltration rate, ponding begins and the boundary condition switches to case (3). Otherwise, the potential infiltration rate is greater than $p$, and $i = p$. Finally, in case (3), the upper boundary condition $H$ is equal to the ponding depth $h$. These cases are schematically illustrated in Figure 2.

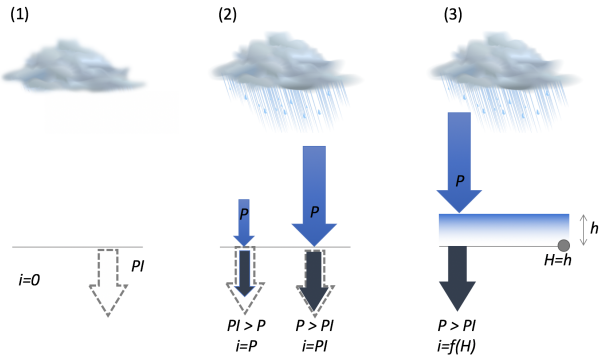


Figure 2: Schematic to illustrate the boundary condition cases for Richards equation. $PI$ is the potential infiltration rate, which is computed to determine whether the rain intensity exceeds the infiltration capacity of the soil (in which case ponding occurs).

# 3 SVE-R model Fortran code

## 3.1 SVE-R model structure

The model is written in Fortran, and located in the file `dry.f`.

Figure 3 summarizes the structure of `dry.f` (see Appendix C for a pseudocode summary).

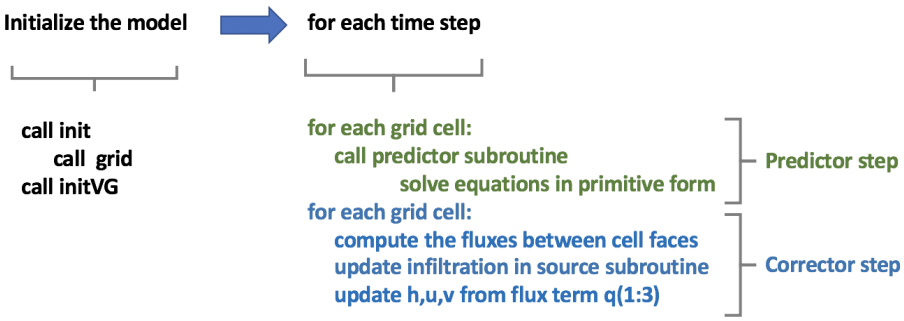


Figure 3: Summary of the SVE-R model structure in `dry.f`.

`dry.f` is initialized with files located in the input folder, and saves all outputs to the output folder. Common variables are specified in `dry.inc` (a number of variables are "common" variables, and not explicitly returned by the Fortran subroutines).

## 3.2 Initializing the SVE solver

The `init` subroutine initializes the SVE component of the model, which involves reading a number of scalar parameters from `params.dat` (see Appendix B) and setting up the grid. The infiltration and parameters are initialized in the subroutine `initVG`). The following subsections include all of the input files, and further details about each input file. The grid-specific input files are illustrated for a 2×2 grid, where the boundaries are: a fixed-flux subcritical boundary at the top of the hill, closed/wall lateral boundaries, and an open boundary at the bottom of the hill. Figure 4 introduces the 2x2 grid used to explain the grid set-up. Note that, while the SVE-R model Fortran code does not assume a rectangular grid, the provided Python code and sample input files do assume a rectangular grid.

### Input files

- `params.dat` : `params.dat` specifies a number of scalar parameters, and is read by the `init` subroutine of `dry.for`. See Appendix for a list of the input parameters and sample `params.dat` file.
- `coords.dat` : contains the $x$,$y$,$z$ coordinates at the cell nodes.
- `boundary.dat`: Describes the boundary types and locations.

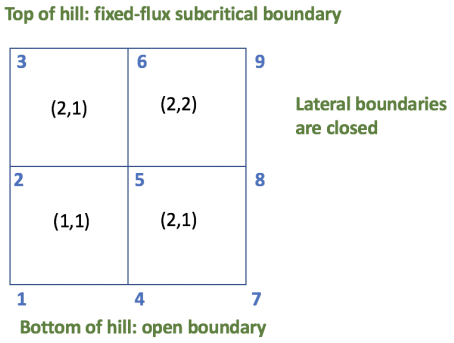


Figure 4: Summary of the SVE-R model structure.

- `veg.dat` : Contains vegetation pattern. Written by `input_veg.py`.
- `nodes.dat` : Numbered list of the nodes surrounding each grid cell.
- `vanG.dat` : Van Genuchten parameters and initial H as a function of depth for vegetated and bare soil cells.

`coords.dat`

`coords.dat` The first line contains the number of points (`npt = (ncol+1)*(nrow+1)`) and the number of cell edges (`ne = nrow*ncol`).

`boundary.dat`

The first row contains the number of boundary cells, defined as a grid cell with one or more boundaries.

columns are `j k inum itype ipos`.

`input_boundary.py` loops over all of the boundary cells, and records the boundary location `j,k`, number `inum`, type `itype`, and position `ipos`. Following this, the following rows record: the number of columns and rows, and, for each column, the beginning and final row (i.e. `kbeg = 1` and `kend = nrow`.

Boundary types are:

- `itype` = 1 for a closed boundary
- `itype` = 0 for an open boundary
- `itype` = 4 for a subcritical fixed flux boundary.

For subcritical fixed flux boundary, the normal depth is computed with Manning's equation with $n = 0.1$. The input parameter `influx` is specified in unis $m^2/s$.

The rainfall rate (in cm/hr) can be related to `influx` as: $p$ (cm/hr) = $q$ ($m^2$/s)*3600 s/hr /(Ly m) * 100 cm/m or $q$*3.6e5/$L_y$

Equivalent `influx` is $p * Ly/3.6e5$ = equivalent $q$ ($\hat{m2}$/s)

### Grid Set-up

- `nn` : space allocated to grid (`nn>np`).
- `np`: the number of nodes.
- `x(nn), y(nn), zz(nn)` : $x, y, z$ coordinates of the cell nodes.
  `x,y,zz` are read as 1D arrays, and interpolated to a 2D grid with the help of the nodes file `nodes.dat`.
- `xc, yc, zc` : coordinates at the cell centers, with dimensions (ncol, nrow)
- `dxi` $= d\xi$, `dxi(1),dxi(2)` $= \xi_x, \xi_y$
- `deta` $= d\eta$ `deta(1),deta(2)` $= \eta_x, \eta_y$
- `sx` $= \frac{dz}{dx}$, `sy` $= \frac{dz}{dy}$
- `nop(nx,ny,4)` : node numbers defining each grid cell (see Figure 5).
- `inum` : number of boundary interfaces of each grid cell.
- `itype` : interface type of each boundary (`itype` = 1 for wall boundaries, 0 for open boundaries,..).
- `ipos` : Position of each boundary, with dimensions (ncol, nrow, 4)

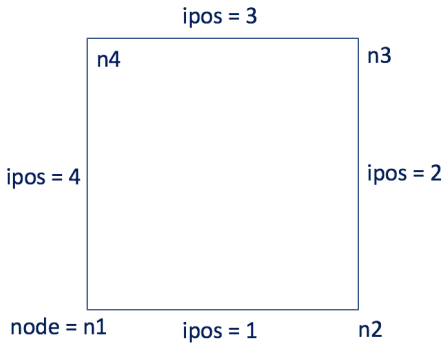


Figure 5: Schematic of the nodes and boundary naming convention.

**2x2 Grid example**

`nop(1,1,:) = [1,4,5,2]` means that cell (1,1) is bounded by nodes `n1=1,n2=4,n3=5,n4=2`, starting from the lower left node `n1` moving counterclockwise. `itype(j,k,2)=1, itype(j,k,3)=4`, means that cell `(j,k)` has a closed boundary at position 2 (right) and a subcritical influx boundary at position 3 (top). See Figure 5.

vertical cell faces have index 1, horizontal faces have index 2

## 3.3 The predictor step

The `predict` subroutine is called for each cell to obtain `hp`, `up`, `vp` (corresponding to $h^{n+1/2}$,$u^{n+1/2}$,$v^{n+1/2}$). `predict` modifies the common variables `hp`, `up`, `vp`, `dh`, `du`, `dv`, `qs`.

The following provides an (incomplete) correspondence between BK equations and Fortran code, with the mathematical notation in *italics* and the equivalent code in `typewriter`. For example, for the array of primitive variables:

$$\mathbf{W_T} = \begin{bmatrix} h \\ U \\ V \end{bmatrix} = \begin{bmatrix} \texttt{h} \\ \texttt{u} \\ \texttt{v} \end{bmatrix}$$

The matrices $\mathbf{A}_W$ and $\mathbf{B}_W$ are defined as:

$$\mathbf{A}_w = \begin{bmatrix} U_\xi & h\xi_x & h\xi_y \\ g\xi_x & U_\xi & 0 \\ g\xi_y & 0 & U_\xi \end{bmatrix} = \begin{bmatrix} \texttt{uxi} & \texttt{h dxi(1)} & \texttt{h*dxi(2)} \\ \texttt{g dxi(1)} & \texttt{uxi} & 0 \\ \texttt{g dxi(2)} & 0 & \texttt{uxi} \end{bmatrix}$$

$$\mathbf{B}_w = \begin{bmatrix} U_\eta & h\eta_x & h\eta_y \\ g\eta_x & U_\eta & 0 \\ g\eta_y & 0 & U_\eta \end{bmatrix} = \begin{bmatrix} \texttt{ueta} & \texttt{h*deta(1)} & \texttt{h*deta(2)} \\ \texttt{g dxi(1)} & \texttt{ueta} & 0 \\ \texttt{g dxi(2)} & 0 & \texttt{ueta} \end{bmatrix}$$

$$\partial_\xi \mathbf{W} = \begin{bmatrix} \partial_\xi h \\ \partial_\xi u \\ \partial_\xi v \end{bmatrix} = \begin{bmatrix} \texttt{dh(1)} \\ \texttt{du(1)} \\ \texttt{dv(1)} \end{bmatrix}; \qquad \partial_\eta \mathbf{W} = \begin{bmatrix} \partial_\eta h \\ \partial_\eta u \\ \partial_\eta v \end{bmatrix} = \begin{bmatrix} \texttt{dh(2)} \\ \texttt{du(2)} \\ \texttt{dv(2)} \end{bmatrix}$$

where `dh(1)` implies `dh(j,k,1)`, and similarly for `du(1)` and `dv(1)`.

$\mathbf{A}_w \partial_\xi \mathbf{W}$ is computed as:

$$\mathbf{A}_w \partial_\xi \mathbf{W} = \begin{bmatrix} U_\xi & h\xi_x & h\xi_y \\ g\xi_x & U_\xi & 0 \\ g\xi_y & 0 & U_\xi \end{bmatrix} \begin{bmatrix} \partial_\xi h \\ \partial_\xi u \\ \partial_\xi v \end{bmatrix} = \begin{bmatrix} U_\xi \partial_\xi h + h\xi_x \partial_\xi u + h\xi_y \partial_\xi v \\ g\xi_x \partial_\xi h + U_\xi \partial_\xi u \\ g\xi_y \partial_\xi h + U_\xi \partial_\xi v \end{bmatrix}$$

Equivalently in code:

$$\mathbf{A}_w \partial_\xi \mathbf{W} = \begin{bmatrix} \texttt{uxi} & \texttt{h dxi(1)} & \texttt{h dxi(2)} \\ \texttt{g dxi(1)} & \texttt{uxi} & 0 \\ \texttt{g dxi(2)} & 0 & \texttt{uxi} \end{bmatrix} \begin{bmatrix} \texttt{dh(1)} \\ \texttt{du(1)} \\ \texttt{dv(1)} \end{bmatrix} =$$

$$\begin{bmatrix} \texttt{uxi*dh(1) + h*(dxi(1)*du(1) + dxi(2)*dv(1))} \\ \texttt{g*dxi(1)*dh(1) + uxi*du(1)} \\ \texttt{g*dxi(2)*dh(1) + uxi*du(1)} \end{bmatrix}$$

The `limitr` subroutine contains the flux limiter that the predictor step uses to compute the cell-average spatial gradients (which preserves solution monotonicity by becoming first-order accurate near discontinuities, yet remains second-order accurate elsewhere). Various choices are included, with the parameter `ilim` specifying the averaging type. `ilim`=5 instructs the code to use the $\beta$ family of averages, and is given by:

$$\overline{\mathbf{\Delta W}} = \text{sign}(a)\text{min}[\text{max}(|a|,|b|), \beta(|a|,|b|)] \text{ if } ab > 0$$
$$0 \text{ if } ab \leq 0$$

$\beta = 1$ yields the relatively more dissipative Minmod average, and $\beta = 2$ yields the less dissipative Superbee average.

## 3.4 Corrector step

The corrector step computes the fluxes between the cell interfaces by calling the `fluxes` subroutine for each cell `j,k` and for each interface.

The interfacial cell fluxes at a given time-step are stored in the common variable `f(0:nx,0:ny,1:3,1:2)`, where the first two indices of `f` contain the $j, k$ coordinates, the third index correspond to the components of $\mathbf{F}_\perp$ (see Equation), and the final index denotes the cell face (1 for vertical cell faces and 2 for horizontal). `f(j,k,1:3,1)` represents the flux from cell $(j-1, k)$ to $(j, k)$, and `f(j,k,1:3,2)` represents the flux from cell $(j, k-1)$ to $(j, k)$ (see Figure **??**). Calling `fluxes(j-1,j,k,k,1)` modifies `f(j,k,1:3,1)` and calling `fluxes(j,j,k-1,k,2)` `f(j,k,1:3,2)`.

### Fluxes subroutine

Within the `fluxes` subroutine, `fluxes(jl,jr,kl,kr,i1)` computes the flux from cell `(jl,kl)` to `(jr,kr)`, modifying the common variable `f(jr,kr,1:3,i1)`. `i1 = 1` indicates that the flux is between vertical cell faces, i.e. from cell `(j-1,k)` to `(j,k)`. Similarly, `i1 = 2` indicates that the flux is between horizontal cell faces, i.e. from cell `(j,k-1)` to `(j,k)`.

`fluxes` first reconstructs the predicted values to the left and right of each face (`hl,hr,ul, ur,vl,vr`) using the monotone upstream scheme for conservation laws (MUSCL). These reconstructed predictor values define a Riemann problem at each cell face, and `fluxes` calls the `solver` subroutine to compute $\mathbf{F}_{\perp \mathbf{I}}$ from Equation **??**, reproduced below:

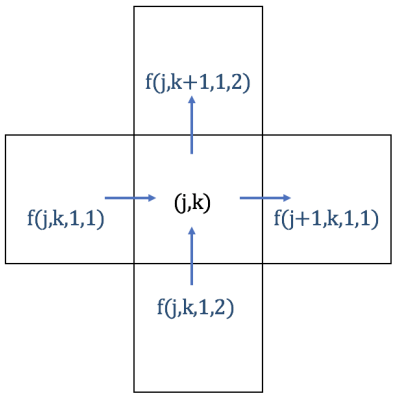


Figure 6: Definitional sketch for the interfacial fluxes, indicating how the fluxes into and out of cell (j,k) are stored in the array `f(0:nx,0:ny,1:3,1:2)`

.

$$\mathbf{F}_{\perp\mathbf{I}} = \frac{1}{2}(\mathbf{F}_{\perp\mathbf{L}} + \mathbf{F}_{\perp\mathbf{R}} - \hat{\mathbf{R}}|\hat{\mathbf{\Lambda}}|\mathbf{\Delta}\hat{\mathbf{V}})$$

**Solver subroutine**

This subroutine uses the monotone upstream scheme for conservation laws (MUSCL) to compute $\mathbf{F}_\perp$ at each cell face. The code in `solver` corresponds to the Equation **??** as:

- $\hat{\mathbf{R}}$ : `e(3,3)`
- $|\hat{\mathbf{\Lambda}}|$ : `a(3)` x
- $\mathbf{\Delta}\hat{\mathbf{V}}$ : `ws(3)`

$$\hat{\mathbf{R}} = \begin{bmatrix} 1 & 0 & 1 \\ \hat{u} - \hat{a}\cos\phi & -\sin\phi & \hat{u} + \hat{a}\cos\phi \\ \hat{v} - \hat{a}\sin\phi & \cos\phi & \hat{v} + \hat{a}\sin\phi \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1 \\ \texttt{uhat-chat*cndum} & \texttt{-sndum} & \texttt{uhat+chat*cndum} \\ \texttt{vhat-chat*sndum} & \texttt{cndum} & \texttt{vhat+chat*sndum} \end{bmatrix}$$

$$|\hat{\mathbf{\Lambda}}| = \begin{bmatrix} |\hat{u}_\perp - \hat{a}| & & \\ & |\hat{u}_\perp| & \\ & & |\hat{u}_\perp + \hat{a}| \end{bmatrix} = \begin{bmatrix} \texttt{|uperp - chat|} & & \\ & \texttt{|uperp |} & \\ & & \texttt{|uperp + chat|} \end{bmatrix}$$

$$\mathbf{\Delta\hat{V}} = \begin{bmatrix} \frac{1}{2}\left(\Delta h - \frac{\hat{h}\Delta u_\perp}{\hat{a}} + \frac{\hat{h}\Delta c_T}{2c_T}\right) \\ \hat{h}\Delta u_\parallel \\ \frac{1}{2}\left(\Delta h + \frac{\hat{h}\Delta u_\perp}{\hat{a}} + \frac{\hat{h}\Delta c_T}{2c_T}\right) \end{bmatrix} = \begin{bmatrix} \texttt{0.5*(dhdum - hhat*duperp/chat)} \\ \texttt{hhat*dupar} \\ \texttt{0.5*(dhdum - hhat*duperp/chat)} \end{bmatrix}$$

- $\Delta h$ : `dhdum = hr - hl`
- Left interface: `hl = hp(jl,kl) + 0.5*dh(i1)`
- Right interface: `hr = hp(jl,kl) - 0.5*dh(i1)`

## 3.5 The `source` subroutine

The source term is computed with the subroutine `source`, which modifies the common variable `qs` (or $Q(j,k)$). `source` is called by both the predictor and corrector steps; however, Richards equation is only solved during corrector step. The input arguments are the cell indices `j,k`, the primitive variables, and an indicator variable `lstep` for the step type (`lstep = 0` for the predictor step and 1 for the corrector step.

In the predictor step, `source` is called for each cell before updating the predictor values (`hp,up,vp`):

```
call source(j, k, h(j,k), u(j,k), v(j,k), 0)
```

`source` is similarly called in the corrector step before updating the fluxes:

```
call source(j, k, hp(j,k), up(j,k), v[(j,k), 0)
```

Inside `source`, the primitive variables are labeled `depth,udum,vdum`. The source term in Fortran code is given as:

$$\mathbf{Q} = \begin{bmatrix} \texttt{winflt} \\ \texttt{-grav*depth*sx(j,k)-grav*depth*fricSx+0.5*udum*winflt} \\ \texttt{-grav*depth*sy(j,k)-grav*depth*fricSy+0.5*vdum*winflt} \end{bmatrix}$$

where $S_{f,x} =$ `fricSx` and $S_{f,x} =$ `fricSy`, and $p - i =$ `winflt`.

#### Infiltration details (corrector-step)

In the corrector step, `source` calls the infiltration-specific subroutines (`timestep` and `potential` to update the infiltration rate $i$ and the surface boundary conditions at cell `j,k`. `timestep` solves Richards equation at cell `(j,k)` and returns the updated $H$, $\theta$ and $K$, which `source` uses to compute the infiltration rate $i$. `potential` is used to prescribe the boundary conditions.

The Richards equation solver is implemented separately for each grid cell, and the 3-dimensional $H$, $\theta$ and $K$ fields are saved as the common variables `r8H` (cm), `r8Theta`, and `r8K` (cm/s), respectively,

with dimensions (`nrow` × `ncol` × `nz`).

From Darcy's law, the infiltration rate $i$ (cm/s) is computed as:

$$i = K\left(\frac{\partial H}{\partial z} + 1\right)$$

which corresponds to `r8kt*((hdum(nz)-hdum(nz-1))/dz+1)` in `source`.

As an intermediate step, the depth is updated with:

```
znew= zold + prate*100*dt - r8kt*( (hdum(nz) - hdum(nz-1))/dz + 1)*dt
```

where `prate` rainfall has been converted to cm/s. `winflt` $= p - i$ is then computed as:

```
winflt = (znew - zold)/dt/100.
```

The surface boundary condition cases are:

- Case 1: Richards equation is solved every `iscale` timesteps. For all other SVE time steps, the infiltration rate is estimated as:

  ```
  r8kt*((hdum(nz) - hdum(nz-1))/dz + 1.d0)*dt
  ```

  if the depth is greater than zero. If depth = 0, q(s)=0, so that $p = i$ between Richards solver updates. Ponding does not occur until the `potential` subroutine determines that the rainfall exceeds to potential infiltration rate ($p > PI$).

- Case 2: Rain and no ponding : `depth=0` and `prate > 0`.
  The potential subroutine is called to estimate a potential infiltration rate $PI$, defined as the infiltration that would occur with $H = 0$ at the surface (in cm/s).
  If the potential infiltration rate is less than the rainfall rate (`(PI .lt. prate*100)`), ponding begins. In this case, the Richards boundary condition is switched to fixed $H$, and the timestep subroutine is called.

## 3.6 `timestep` The Richards equation subroutine

Richards equation is solved by the `timestep` subroutine, which is called from `source`.

### timestep inputs and outputs

The inputs are `hnp1m, thetan`, which are the initial conditions to the Richards eqn solver (`hdum,thetadum` in the source subroutine, which calls `timestep`). The outputs are `hnp1mp1,r8thetanp1m,r8knp1mp1`, which are $H$, $\theta$ and $K$ at the following timestep (after dt_r time elapsed).

When the soil is flagged as saturated, the surface flux is set to $K$ at the surface, which should be very close to saturated (`flux = - r8knp1m(nz)`). This is achieved by adjusting the value of $H$ at node (nz-1):

```
hnp1mp1(nz-1) = hnp1mp1(nz) + dz + flux*dz/r8knp1m(nz-1)
```

Note: this approach should be treated with caution. Is was designed for the use case of a fixed intensity rain storm, in which a uniform wetting front would arise. It is not meant for variable rainfall cases.

#### Stop tolerance

`stop_tol0` is the stop tolerance specified in the input `params.dat`, with default value `stop_tol0 = .01`. `stop_tol` is reset to the original `stop_tol0` at the beginning of each Richards solver timestep. The convergence tolerance is relaxed if the Richards solver fails to converge within a specified number of iterations (default = 100).

#### Potential infiltration

The subroutine `potential` is used to compute the potential infiltration rate $PI$, defined as the infiltration that would be observed for a surface boundary condition of $H = 0$. Potential infiltration is computed when there is rain but no ponding, and returns the potential infiltration rate, defined as the infiltration rate that would occur with $H = 0$ at the surface.

# 4 Running the model

The Fortran code can be compiles from the terminal as `gFortran -o ./sw -framework accelerate ./dry.for` and executed as `.sw`, where `sw` is the name of the compiled executable.

#### input_coords.py / wrap_coords

`sim_input.py` calls `wrap_coords` for each path / parameter case:

```
wrap_coords(path + '/input', params)
```

which calls the following:

- `write_nodes(path, ncol, nrow)` : writes the cell node indices to `input/nodes.dat`. This function assumes that the grid is rectangular, and would need to be modified for a non-rectangular grid. It saves `nop`, which contains the indices of the nodes surrounding each cell face.
- codebuild_coords(params) : constructs the x,y,z fields.

## 4.1 Output files

Output files are read by Python script `batch_read.py`, using functions from `output_dry.py`. Most outputs are saved at intervals of $dt\_p$, to limit the size of the output files.

- `time.out` : file containing time, and max CFL number at each timestep.

- `h.out` : contains $h$,$U$, $V$,
- `summary.out` : summarizes ponding time, runtime, final time, and if applicable, the reason for an early exit (i.e. no more water, AMAX too big)
- `hydro.out` : hydrograph at 1 second time resolution. The hydrograph is saved by Fortran with $m^3/s$ units, and normalized by the hillslope dimensions to convert to cm/s in Python. Read by function `new_hydro.out` in `output_dry.py`.
- `dvol.out` : output file to check mass balance. Read by function `get_dvol` in `output_dry.py`. See details below.
- `ptsTheta.out` (optional) : evolution of the soil moisture profiles at two points, one vegetated and one bare.

**h.out**

Columns are

1. `j,k,h,u,v,zinflmap2, xflux0,yflux0,xflux1,yflux1`
2. Rhese variables are written for every grid cell in the domain (every j,k) every nprt timesteps. Following, a line
3. These variables are written to file when `myoutput` is called, nprt time steps.
4. `zinflmap2` contains the total volume of water infilrated in cell j,k, and is reset to zero as soon as this variable is written to file.

The Python function `get_h` in `output_dry.py` reads `h.out` and returns: `h,u,v,inflVmap,xflux0,yflux0,xflux1,yflux1` all with dimensions: `nprt x ncol x nrow`. Units are:

- `h` : m
- `u, v` : m/s
- `inflVmap` : cm $m^2$ (converted from Fortran $m^2$ )
- `xflux0, yflux0, xflux1, yflux1` : cm $m^2$ (converted from Fortran $m^2$ )

  fluxes are positive directed out of the domain.

`zinflc`: infiltration depth in cm

**dvol.out**

Contains output from Fortran mass balance tracking. Columns are `dvol, flux, infl,zain` from Fortran mass tracking, containing the change in surface volume, horizontal fluxes out of the domain, infiltration and rain volumes, respectively, in units $m^3$.

The Python function `get_dvol` in `output_dry.py` reads only the first three columns dvol, flux, infl, and the last is available for mass balance debugging. `get_dvol` normalizes by the domain area (Lx*Ly) and converts to cm.

# 5 Python wrappers

Python scripts generate the Fortran input files and read the Fortran output files.

- `call_dry.py`
  - `sim_input`
  - `runmodel`
  - `read_sim`

`call_dry.py` is a control script which writes the Fortan input files, compiles and executes the Fortran code dry.for, and reads the output files.

`sim_input` makes the input files.

## 5.1 sim_input

`sim_input` specifies default parameters where no parameters are provided in `params.json`. For example, if `ncol` is not provided, an aspect ratio must be included instead (`aspect`)

Notes on roughness parameters:

- roughness parameters are specified separately for vegetated and bare cells.
- parameters for the resistance formula for vegetated cells are specified as `alpha`, `eta` and `m` ($\alpha$, $\eta$ and $m$), and for bare soil cells as `alpha_B`, `eta_B` and `m_B` ($\alpha_B$, $\eta_B$ and $m_B$).s
- The default soil parameters are Manning's equation with `alpha = 0.1`, and `alpha_B = 0.03`.

Some variables cannot be initialized / specified from `params.json` and must be modified within the code:

- inflow boundary dimensions (currently = entire upslope area)
- van Genuchten parameters of the seal layer
- van Genuchten parameters are hard coded be the same between vegetated and bare cells (this can be changed in `input_phi.py`).

Note on the inflow boundary condition: If the upslope boundary is a fixed flux boundary, `tr` also specifies that the time as which the inflow stops (i.e. the boundary changes from fixed flux to a wall boundary). In the case of rain AND upslope inflow, for example, to simulate rain on a longer hillslope domain, the rain and inflow will end at the same time. The code would need to be modified with the addition of a new variable (e.g. time of inflow end) to change this, which would be straightforward.

#### General parameters

#### SVE parameters

- `dx` : grid discretization (default = 1.)

- `nrow` : Number of grid cells in the longitudinal (along-slope) direction (default = 40).

  Hillslope length `Ly` = `nrow * dx`

- `ncol` : Number of grid cells in the transverse (across-slope) direction (default = 10).

  Hillslope width `Lx` = `ncol * dx`

- `topo` : default = "plane", specifying a planar hillslope, aligned along the y-axis. other option is to specify coordinates.

- `vegtype` : specified how the vegetation field is built

  `randv` : random vegetation field, defined by $\sigma_V$ and $f_V$

  `image` : read from image file.

- `scheme` : resistance formulation (default in Manning equation)

  specifying the scheme is equivalent to specifying eta and m (see source subroutine, where roughness formulation is implemented. Explicitly specifying m and/or eta will overwrite these default parameters.

- `alpha` / $\alpha_V$ : generalized roughness parameters for vegetated / permeable areas. If Manning's equation is used to specify the resistance formulation, then $\alpha_B = n_B$.

- `alpha_B` / $\alpha_B$ : equivalent to `alpha` / $\alpha_V$ for bare soil areas.

- `epsh` / $\epsilon$: depth tolerance for dry cells. The momentum equations are not solved if `h< epsh` ($h < \epsilon$).

- `beta` / $\beta$ : a constant in limiter subroutine if ilim = 5, beta family (default = 1).

**Infiltration parameters**

- `KsV` : infiltration rate in vegetated/permeable areas, in cm/hr.

  Note `ksatV=KsV/3600.` is the infiltration rate in units of cm/s.

- `KsB` : infiltration rate in bare soil/impermeable areas, in cm/hr.

  Note `ksatB=KsB/3600.` is the infiltration rate in units of cm/s.

  If `KsB` is not specified, then `KsB` is set to `KsV/s_scale`, where `s_scale` is the ratio.

- `s_scale` : the soil infiltration ratio between vegetated and non-vegetated (or permeable/impermeable) areas.

- `stop_tol` : error tolerance for Richards equation convergence, default = 0.01

- `tsat_min`: default = 600. Specified in seconds. Minimum time until the soil is allowed to saturate ($i$ is set to $K_{sat}$, bypassing the Richards equation solver.

- `H_i` : initial $H$ at the surface. The soil is initialized to an equilibrium profile (this can be modified in `sim_input.py`.

- `tr` : storm duration (min)
- `p` : rain intensity (cm/hr)
- `rain` : rain intensity (mm/s)
- `isvegc` : veg field (1 for permeable areas, 0 for impermeable)

### 5.2 To write Fortran input files: `sim_input.py`

**Input vegetation file**

`sim_input.py` calls `input_veg.py` to write the input vegetation file (`veg.dat`) Vegetation types

- `randv` :
- `image` :

## 6 Template iPython notebook files

Template notebooks:

- notebooks to read the output of the SVE-R model
- GW : stand alone implementations, and code to compare to SVE-R model output.
- Richards equation : stand alon, basic and module versions
- Code to compare simulation output to the kinematic wave approximations –¿ related to friction sims.
- Code to compare $IF$ predictions by KWA, SVE-R and GW.
- roughness matching

Stand alone implementations

### 6.1 SVE-R model

**Template notebooks to process / view the SVE-R simulation results:**

- `illustrate.ipynb`: Illustrate simulation results
- `illustrate_3D`: Illustrate simulation results in 3D. Useful for visualizations.
- `illustrate_soils.ipynb`: Plot soil infiltration profile.
- `mass_balance_check.ipynb`: Check mass balance in the SVE-R model.
- `boundary_flux.ipynb`: Check lateral fluxes in the SVE-R model.

**Template notebooks to view / modify how the SVE-R inputs are constructed**

- `input_coords.ipynb`: Illustrate how the topography is created. Includes limited 3D plotting functionality. Case: planar...
- `input_veg.ipynb`: Illustrate how the vegetation is created. Case: randv, stripes (random widths and spacing), read image.
- `image_read.ipynb` : Code that reads image file and adjusts to the specified hillslope dimensions. After `wrap_image_veg` function in `input_veg.py`.

## 6.2 Giraldez and Woolhiser

- `Giraldez_Woolhiser.ipynb` : Basic implementation of the method of characteristics.
- `GW_check.ipynb` : Basic code to compare SVE-R simulations to GW prediction, for a single simulation.
- `mass_balance_with_GW` : Compare GW and SVE-R predictions of $IF$, rising and recession time (loop over all simulations).

  Assumes Manning's roughness

## 6.3 Roughness

- `scheme_compare.ipynb`
- `scheme_compare.ipynb`

## 6.4 Richards stand-alone code

1. Richards solver implementation in Python (to assess sensitivity to Richards discretization)

   A stand alone Python implementation is available in the jupyter notebook file `Richards.ipynb` and Python module `richards.py`.

2. GW code (to compare SVE-R and GW solutions. note, also includes code to estimate soprtivity from Van Genuchten parameters)

**Richards alone ipynb notebooks**

- `Richards_basuc.ipynb` : a simple ipynb implementation of the Richards equation solver, following *Celia et al.* (1990). Translated from iowa matlab code:
  `https://leaf.boisestate.edu/blog/2010/11/07/richards-solver-code/`
- `Richards.ipynb` : uses `Richards.py` from `richards1D.py`. Need to add crust functionality and saturation logic to this module!f

•

The stand-alone 1D Richards equation solver is structured as a class `Richards`, which takes as input a parameter dictionary `params`.

# 7 Appendix A: `dry.f` parameters and variables

## 7.1 Source subroutine variables

- `isetflux` (common) specifies the boundary condition for the Richards equation solver.

  `isetflux` = 1 if the surface flux is specified: `flux = 0` in case 3 (no rain and no ponding), and `flux = prate` in case 2 (rain and no ponding).

  `isetflux` = 0 if the surface H is specified (Dirichlet). If there is ponding (case 4), the surface boundary condition is $H$=$h$.

- `flux` is the surface flux (cm/s). `flux` is a common variable defined in `source`, and used by `timestep` if `isetflux=1`.

- `htop`: Upper boundary condition for fixed $H$ boundary condition (`isetflux=1`).

- `PI` Potential infiltration, computed by the `potential` subroutine.

- `depth, udum, vdum` : $h$, $U$, $V$ at grid cell `j,k`.

  `hp(j,k),up(j,k),vp(j,k)` if called from the corrector step.

  `h(j,k),u(j,k),v(j,k)` if called from the predictor step.

- `lstep` indicates from which sstep the source subroutine was called.

  `lstep` = 0 if called from the predictor step

  `lstep` = 1 if called from the corrector step.

  Infiltration is only updated depth in corrector step.

- `zold, znew` : depth in cm

  Richards solver units are in cm, whereas SVE-R solver units are in m)

- `fm, feta, falpha`: roughness parameters $m$, $\eta$, and $\alpha$.

# Appendix B: dry.f input files

Example input files are included here for the 2x2 grid example, to illustrate. Figure 7 shows the cell center and node indices for the example grid, and Figure 8 shows the boundaries.

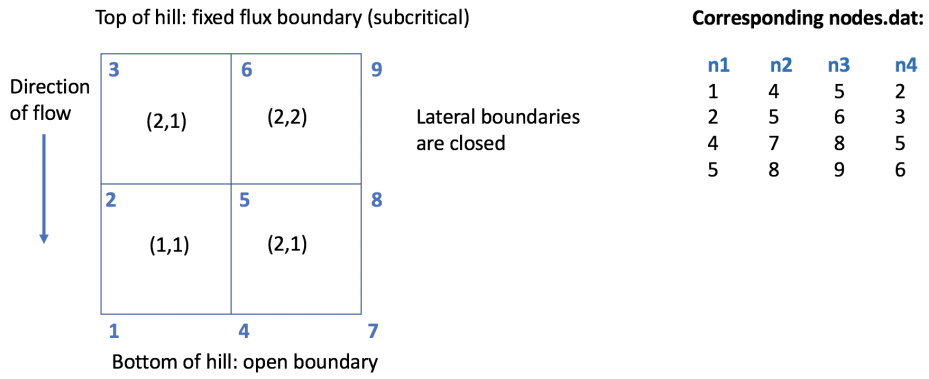


Figure 7: Example 2×2 grid to illustrate how cell centers and nodes are assigned. (j,k) coordinates are shown in the cell centers, and the node indices are at the cell corners.

## 7.2 params.dat

Sample `params.dat` files can be found in the `input` directories of the examples. The parameters specified in `params.dat` are:

- `grav` : acceleration due to gravity.
- `dt` : SVE timestep (`dt` is labeled `dt_sw` in the Python code).
- `tmax` : maximum time until the simulation ends.
- `tr` : rain duration in seconds (`tr` is labeled `t_rain` in the Python code).
- `prate` : rain intensity in m/s.
- `nt` : number of time steps (`tmax/dt`)
- `epsh` : depth threshold to solve the SVE momentum equations.
- `beta` : value of beta in limitr subroutine.
- `nprt` : frequency f which the SVE-R output fields are saved: `nprt= dt_p`. Note: the hydrograph is saved every second.
- `iscale` : ratio of SVE to Richards equation timestep durations.
- `stop_tol` : convergence criteria for Richards equation solver (default = 0.01).
- `h0, u0, v0` : initial depth, `u`, and `v` (defaults = 0).
- `r8mV,r8etaV, r8alphaV` : roughness parameters $m$, $\eta$ and $\alpha$ for vegetated cells.
- `r8mB,r8etaB, r8alphaB` : roughness parameters $m$, $\eta$ and $\alpha$ for bare cells.
- `jveg, kveg` : `j, k` indices of a vegetated cell for which the soil profile ($H$ and $\theta$) is saved every `nprt` timesteps.

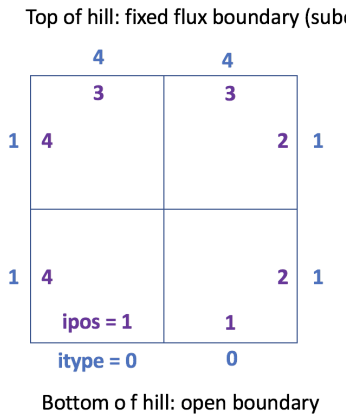


Figure 8: Boundaries for the 2×2 grid in Figure 8. The boundary types are labeled in blue outside the grid, with 0 for the open boundary, 1 for the closed boundaries and 4 for the fixed flux boundary.

- `jbare, kbare` : `j, k` indices of a bare soil cell for which the soil profile ($H$ and $\theta$) is saved every `nprt` timesteps.
- `tsat_min` : Minimum time until the soil is allowed to "sarturate" (default =0).

## 7.3 boundary.dat

Sample `boundary.dat` for the 2x2 grid example, for which every cell is a boundary cell. The array on lines 3-7 show the (j,k) indices for each boundary cell, the number of boundaries associated with that cell, and boundary type of each boundary cell.

```
number of boundary cell
  4
 j     k          inum     itype              ipos
1      1             2        1         0          4         1
1      2             2        1         4          4         3
2      2             2        4         1          3         2
2      1             2        1         0          2         1
ncol
2
nrow
2
j      kbeg          kend
    1      1             2
    2      1             2
number of fixed bc cells, ndir
 2
 j     k     fix h     fix u     fix v
```

```
1     2     0.0     0.0     -2e-05
2     2     0.0     0.0     -2e-05
```

### 7.4 coords.dat

In the first row, the first value specifies the number of nodes (`npt`), and the second specifies the number of cells (`ne`). The remaining rows list the x,y,z coordinates at the nodes, in order of node number. Below is an example `coords.dat` file for the 2×2 exampleg rid.

```
9       4
0.0     0.0     0.0
0.0     1.0     0.02
0.0     2.0     0.04
1.0     0.0     0.0
1.0     1.0     0.02
1.0     2.0     0.04
2.0     0.0     0.0
2.0     1.0     0.02
2.0     2.0     0.04
```

### 7.5 coords.dat

## Appendix C : `dry.f` pseudocode

### 7.6 Model overview/summary

---

```
call init ! grid subroutine called from the input subroutine
call myoutput ! write initial conditions to file


do it = 0,nt-1 ! loop over time steps (it = time iteration number)
   t = t+dt  ! increment time (t = current time)

   ! Predictor step
   loop over grid cells:
      call bconds(j, k, h, u, v) ! update h,u,v values in ghost cells
      call predict(j, k)         ! get predictor variables hp, up, vp

    ! Corrector step
   ! Corrector part 1: compute the fluxes between cell faces
   loop over grid cells:
    call bconds(j,k,hp,up,vp) ! update hp,up,vp in the ghost cells
    ! compute interfacial fluxes.
    call fluxes(j-1,j,k,k,1)   ! vertical faces.
```

```
    call fluxes(j,j,k-1,k,2)   ! horizontal faces.
   loop over boundaries:
      compute boundary fluxes

   ! Corrector part 2:
   loop over cells:
      call source ! update the source term qs
      compute q from qs and f ! q = (h,uh,vh)
      ! update h,u,v from q. check
       if q > 0:          ! check for negative depth.
         h = q(1)          ! if positive depth, update h
       else:
         q = 0            ! if negative depth, reset h and q(1) to zero
         h = 0

       if h < epsh:    !  neglect momentum in nearly dry cells
         u, v = 0.     !  set u and v to 0 in nearly dry cell
         q(2:3) = 0    !  set momentum fluxes to 0

       elif h >= epsh:    !  store u and v values
         u, v = q(2)/h, q(3)/h

    ! Exit when the ponded water is less than min_vol
    if volume < min_vol
      call gracefulExit

    ! Exit when the CFL number is too big
    if(CFL > 10) then
        call gracefulExit

   ! Exit if time runs out
    if(t > tmax) then
        call gracefulExit
```

\end{verbatim}

\subsection{Grid pseudocode}
Pseudocode for grid setup:

\begin{lstlisting}

```
subroutine grid
include "dry.inc" ! include common variables
read np, ne   ! np: number of grid points, ne : number of cells
read x, y, zz ! from coords.dat, coordinates of the cell nodes
read veg      ! from veg.dat, vegetation at the cell nodes
read nop   ! from nodes.dat, the node numbers surrounding each cell.
```

```
! Compute grid metrics.
for each cell (j,k):
   n1, n2, n3, n4 = nop(j,k,1:4) ! get the node numbers
   ! compute xc, yc, zc as the average of x,y, zz at the surrounding nodes
   xc(j,k) = 0.25*(x(n1) + x(n2) + x(n3) + x(n4))

   compute dxi, deta, area ! compute grid metrics
   compute sx, sy, dz, ds, sn, cn

loop over cells: ! Set values in the ghost cells
   loop over faces:
      call findbc(i,j,k,jj,kk,j2,k2) ! get ghost cell indices (jj, kk)
      set sx, sy, dxi, deta in ghost cell equal to values in boundary cell (j,k)
```

## 7.7 Source pseudocode

```
! select the soil profiles from the r8H, r8Theta, r8K
hdum = r8H(j,k,1:nz)
thetadum = r8THETA(j,k,1:nz)
r8kdum = r8K(j,k,1:nz)

! convert the ponded depth to cm to use with as the Richards equation BC
zold = depth*100.d0 ! input depth in cm
znew = zold    ! initialize depth after the infiltration step in cm
PI = 0.d0    ! Initialize the potential infiltration as 0.
iskip = 0    ! Initialize the infiltration indicator (iskip = 1 if Richards solver is
    skipped, in which case r8H, r8Theta and r8K matrices are not updated).

! determine whether the cell is vegetated or not, and define the roughness parameters
    accordingly:
if vegetated cell:
    isveg = 1
    fm,falpha,feta = r8mV, r8alphaV, r8etaV
else: ! bare cell
   isveg = 2
    fm,falpha,feta = r8mB, r8alphaB, r8etaB

! only update depth in the corrector step
if lstep = 1 then

! Before the Richards equation solver is called, several cases need to be handled.
! only solve Richards equation every iscale timesteps
   if mod(it, iscale) != 0 then
     if there is ponding, set the the surface flux / infiltration rate using K from the
         previous timestep
! Case 2: rain and no ponding
   else (if prate > 0 and zold = 0) then
```

```
    call potential(hdum, thetadum, PI) ! determine the potential infiltration rate PI. Note:
        potential is a subroutine that takes hdum and thetadum as inputs and returns PI as
        the output.
!   Handle two cases: PI > prate (no ponding) and PI < prate (ponding starts)
    if  prate > PI then
        record the time of ponding, tp
       isetflux = 0  ! set the Richards boundary type to fixed H (Dirichlet), with htop=0 at
           the surface.
       call Richards equation with the updated boundary conditions.
    else   ! no ponding
      isetflux = 1
      flux = - prate*100. ! set the infiltration  rate to the rainfall rate
       winflt = (znew - zold)/dt/100.d0
! Case 3: no ponding and no rain
   else if (zold = 0 and prate = 0) then
      isetflux = 1 ! set the Richards surface boundary condition to fixed flux
     flux = 0    ! set flux to zero
     call Richards equation solver to update the soil moisture profile
     winflt = 0
! Case 4: ponding (with or without rain)
  else if zold > 0 then
      isetflux = 0 ! set the Richards surface boundary condition to fixed H
     htop = zold    ! set surface BC to ponded depth
     call Richards equation solver with updated boundary conditions.

      update depth (znew) using the infiltration rate from the updated K
      winflt = (znew - zold)/dt/100.d0
! End of Richards equation solver cases.

  if iskip = 0 ! if Richards equation was called by timestep
       ! update r8THETA, r8H, r8K (3D soil matrices) with 1D solver results
       r8THETA(j,k,1:nz) = thetap1dum
      r8H(j,k,1:nz) = hp1dum
      r8K(j,k,1:nz) = r8kp1dum

       compute r8fluxin (surface flux)

       if ipass = 0 ! if Richards equation was not passed (due to saturation flag)
       compute r8fluxout ! drainage flux
     else ! saturation flag went off
         r8fluxout = r8fluxin ! assume soil flux out = flux in
     compute r8newmass ! change in soil moisture since the previous timestep

     ! update oldTHETA (soil moisture content from the previous timestep)
     oldTHETA(j,k,1:nz) = r8THETA(j,k,1:nz)
else  ! in the predictor step, there is no infiltration or precipitation
  winflt = 0.d0

if (depth > epsh) then ! depth greater than threshhold
      ! compute magnitude
```

```
      vmag = dsqrt(udum*udum + vdum*vdum)

      if (feta .eq. 0.5) then ! non-laminar schemes

         fricSx = (falpha/depth**fm)**(2.d0)*udum*vmag
         fricSy = (falpha/depth**fm)**(2.d0)*vdum*vmag


      elseif (feta .eq. 1) then ! special case for laminar

         fricSx = falpha*udum/depth**fm
         fricSy = falpha*vdum/depth**fm

         ! include a catch for high Re cases: use DW ff with f = 0.5
         Rel = vmag*depth/1.e-6

         if (Rel .gt. 500) then
             ffact = 0.5 ! okay for smooth surfaces (following Kirstetter)
             fricSx = ffact*udum*vmag/8./grav/depth
             fricSy = ffact*vdum*vmag/8./grav/depth
         endif

       endif

      ! update the source terms
       qs(1) = winflt

       qs(2) = 0.5D0*udum*winflt - grav*depth*fricSx -
     &         grav*depth*sx(j,k) ! sx(j,k) = x-dir bed slope
       qs(3) = 0.5D0*vdum*winflt - grav*depth*fricSy -
     &         grav*depth*sy(j,k) ! sx(j,k) = y-dir bed slope
      else

       qs(1) = winflt
       qs(2) = 0.d0
       qs(3) = 0.d0

      endif
```

## 7.8 `timestep` pseudocode

Pseudocode for subroutine `timestep`:

```
 subroutine timestep(hnp1m,thetan,hnp1mp1,r8thetanp1m,r8knp1mp1)
!
!   Input:
!            hnp1m, thetan (real, kind = 8) - initial
!   Output:
```

```
!           hnp1mp1,r8thetanp1m,r8knp1mp1 - h, theta and k at time m+1
!
!  Comments: uses common variables nz, stop_tol, htop as surface h
!            modifies common variable ipass
  include 'dry.inc'


  declare variables hnp1m(nz), thetan(nz), r8cnp1m(nz), r8knp1m(nz) ... ! input and output
      arrays, and arrays used by the Richards eqn solver
  istop_flag = 0 ! indicator variable switches to 1 when convergence criteria is met
  niter = 0 ! number of iterations
  stop_tol = stop_tol0 ! reset stop tolerance to input (in case condition was relaxed)
  ipass = 0 ! indicator variable, set to 1 if soil is 'saturated'

  do while stop_flag = 0:
     ! r8cnp1m,r8knp1m,r8thetanp1m given hnp1m
     call vanGenuchten(k,hnp1m(k),
         r8cnp1m(k),r8knp1m(k),r8thetanp1m(k), isveg)

   Do some linear algebra (see Celia et al. (1990)

   ! Compute deltam, the increment in iteration for iteration m+1
   deltam = matmul(Ainv, R_MPFD)

   increment niter (number of iterations)
   niter = niter + 1

   if niter > 100
      stop_tol = stop_tol*10 !relax stop tolerance
      if stop_tol > 10:
         quit and return error

   ! Handle saturation cases
   compute t2b_theta = theta(top) - theta(bottom) ! the soil moisture difference between the
       top and bottom of the soil profile.

   ! Test whether the soil has saturated
   ! After time tsat_min, if there's ponded water at the surface:
   ! If the soil moisture difference between the surface and bottom is very small, then:
   if t > tsat_min and depth > 0
     if t2b_theta < 0.005, then
       hnp1mp1, thetanp1m = hnp1m, thetan ! update h and theta to (n+1, m+1)
       flux = - r8knp1m(nz)  ! set flux to K(surface)
       ! adjust hnp1mp1(nz-1) to obtain this flux
       hnp1mp1(nz-1) = hnp1mp1(nz) + dz + flux*dz/r8knp1m(nz-1)

       ipass = 1   ! flag the soil as saturated
         istop_flag = 1 ! exit Richards solver - don't wait for convergence
   ! Give the soil a chance to unsaturate after the rain, and the water has drained
   ! Test whether a saturated soil has unsaturated
```

```
! After the storm, if there's no ponding
if t > tr and depth = 0
  if t2b_theta < 0.005, then !if the soil moisture difference between the surface and
      bottom is very small, then the soil was flagged as saturated.

   hnp1mp1, thetanp1m = hnp1m, thetan ! update h and theta to (n+1, m+1)

   flux = 0   ! set surface flux to zero

   ! set hnp1mp1(nz-1) to achieve zero flux BC at the surface
   hnp1mp1(nz-1) = hnp1mp1(nz) + dz
   ipass = 0  ! unfreeze the soil
   isetflux = 1 ! set a fixed, zero flux boundary condition


if (ipass eq 0) then ! if the soil is not saturated then...

   if max(deltam) < stop_tol then ! convergence criteria has been met
      istop_flag = 1

         hnp1mp1 = hnp1m + deltam ! update h(n+1,m) to (n+1,m+1)

         ! apply surface boundary conditions
         if (isetflux .eq. 0) then ! fixed H
           hnp1mp1(nz) = htop
         elseif (isetflux .eq. 1) then ! fixed flux
           r8gkt = r8knp1m(nz-1)
           hnp1mp1(nz) =  hnp1mp1(nz-1) - dz - flux*dz/r8gkt
         endif
         ! apply free drainage BC at the lower boundary
         hnp1mp1(1) = hnp1mp1(2)

         call vanGenuchten

      else  ! update h and keep iterating

         hnp1mp1 = hnp1m + deltam ! update h(n+1,m) to (n+1,m+1)
         hnp1m = hnp1mp1 ! update the old h(n+1,m)

      ! apply surface and lower boundary conditions
```